C# e Visual Studio

Desenvolvimento de aplicações desktop

EVERTON COIMBRA DE ARAÚJO

Introdução

Como sempre digo na abertura de todos os meus livros, ensinar e aprender são tarefas que andam juntas e, para que seja alcançado o objetivo em cada uma delas, é necessário muita dedicação e estudo constante. Não há mágica no aprendizado, mas há muita determinação por parte daquele que quer aprender.

Este livro apresenta um pouco do que o .NET pode oferecer em aplicações voltadas ao desktop, fazendo uso da linguagem C# e do Visual Studio como ambiente (IDE – *Integrated Development Environment*) para o desenvolvimento. Cada recurso apresentado aqui é conceituado e exposto por meio de exemplos práticos, os quais são trazidos com o objetivo de aguçar a sua curiosidade.

Certamente, este livro pode ser usado como uma grande ferramenta em disciplinas que trabalham o desenvolvimento para ambientes gráficos e integrados (IDE), quer seja por acadêmicos ou professores. Isso porque ele é o resultado da experiência que tenho em ministrar aulas dessa disciplina, então trago para cá anseios e dúvidas dos alunos que estudam comigo.

O objetivo deste trabalho é torná-lo uma ferramenta presente e constante no ensino e aprendizado no desenvolvimento de aplicações para desktop, fazendo uso de C#, mais precisamente e de forma direcionada às disciplinas ministradas em cursos que envolvam informática e computação, como Sistemas de Informação, Ciência da Computação, Processamento de Dados, e Análise e Desenvolvimento de Sistemas. Para isso, neste livro trago definições e observações referentes às partes que compõem o desenvolvimento de aplicações desktop e da linguagem C#. Todo o desenvolvimento é realizado com auxílio do Visual Studio 2013.

Talvez surjam dúvidas em relação às versões do IDE, mas o que abordamos aqui é o básico para você se iniciar na programação para ambientes visuais. Aliás, não será obstáculo algum caso você tenha acesso a uma versão diferente (anterior ou superior) do que a trabalhada. Isso porque os conceitos básicos apresentados e necessários que você tenha estão disponíveis em todas as versões existentes.

O livro é todo desenvolvido em oito capítulos, todos com muita prática, e de uma conclusão dos tópicos vistos. Na sequência, são apresentados pequenos resumos do que é trabalhado em cada um destes capítulos.

O repositório com todos os códigos-fonte utilizados no livro podem ser encontrados em: https://github.com/evertonfoz/livro-windows-forms-casa-do-codigo. Estou trabalhando em um blog, como laboratório para meu próximo livro. Acesse http://evertoncoimbra.wix.com/asp-net-mvc-e-ef e contribua.

Capítulo 1 – As primeiras implementações: estrutura sequencial

Os três primeiros capítulos deste livro aplicam as estruturas básicas que toda linguagem oferece: sequencial, condicional e de repetição. Dessa maneira, neste primeiro capítulo são apresentados problemas que podem ser resolvidos fazendo uso de uma estrutura sequencial. Durante as implementações propostas, princípios e fundamentos relacionados à plataforma .NET, à linguagem C# e ao Visual Studio também são trabalhados e apresentados. Todos as soluções serão baseadas em uma interface gráfica com o usuário, nomeada na plataforma como *Windows Forms Application*.

Capítulo 2 – Evoluindo na complexidade: estrutura condicional

Dando sequência, este capítulo apresenta novos problemas que podem ser resolvidos por meio da aplicação de estruturas condicionais, também conhecidas como estruturas de seleção. Para essas implementações, busca-se apresentar novos recursos para aplicações Windows Forms, assim como uma evolução na linguagem C#, com novos recursos e técnicas. Um desses recursos e técnicas refere-se à validação de valores informados pelo usuário, nos controles que permitem interação.

Capítulo 3 – Resolvendo problemas mais complexos: estrutura de repetição

Existem situações nas quais um problema exige ou apresenta como resolução um conjunto de instruções que devem ser executadas por mais de uma vez. Para essas situações, é feito o uso de uma estrutura de repetição que, assim como a condicional, possui variações de sua implementação. Dessa maneira, este terceiro capítulo apresenta a última das três estruturas básicas para uma linguagem de programação, trazendo problemas que poderão ser resolvidos com o uso desse tipo de estrutura, usando suas variações. Novos controles e recursos também são apresentados, como a leitura e escrita de um arquivo texto, e controles para interação com conjuntos de dados. Também, a aplicação de Orientação a Objetos (OO) começa a ser mostrada neste capítulo.

Capítulo 4 – Vários projetos em uma solução: aplicando conceitos relacionados ao MVC e Orientação a Objetos

Nos capítulos iniciais, a preocupação foi no conhecimento dos recursos básicos para desenvolvimento de aplicações Windows Forms, da linguagem C# e do Visual Studio, onde os projetos criados eram independentes. Desta maneira, este capítulo apresenta recursos, conceitos e técnicas mais avançados, possibilitando a criação de uma solução com vários projetos, delegando responsabilidades a cada um deles e tornando a solução uma única aplicação. Nessa linha, é apresentado o MVC (Modelo-Visão-Controle ou *Model-View-Controller*), um modelo de desenvolvimento – que pode ser visto como um padrão – que permite que cada camada (no exemplo, cada camada refere-se a um projeto) realize especificamente o que é de sua responsabilidade. Com vistas à implementação de uma solução em MVC, técnicas relacionadas a Orientação a Objetos são apresentadas, assim como o uso de coleções (*collections*) e associação entre as classes.

Capítulo 5 – Acesso a dados por meio do ADO.NET

Até o quarto capítulo, os dados utilizados pela aplicação são obtidos e fornecidos no mesmo momento em que ela é executada, não havendo possibilidade de recuperar futuramente os dados informados em uma execução

anterior. Uma exceção para essa situação se dá no capítulo 3, onde é possível persistir os dados em um arquivo de textos e recuperá-los em qualquer execução futura da aplicação. Entretanto, normalmente a persistência de dados é realizada em uma base de dados, e o .NET oferece toda uma arquitetura para essa funcionalidade, que é o ADO.NET. Aqui, são apresentados recursos e ferramentas oferecidos pela plataforma, e o Visual Studio que permitam a persistência e recuperação de dados em uma base de dados, permitindo aplicar todo o conteúdo já apresentado até este capítulo. O banco de dados usado neste livro é o SQL Server Express Edition.

Capítulo 6 – Utilizando DataSet tipado para acesso a base de dados

Com o conhecimento do ADO.NET, apresentado no Capítulo 5, no qual o acesso a dados é realizado por meio de classes da API e instruções SQL, é possível saber como a interação com o banco de dados ocorre. Este capítulo avança e apresenta que, para acessar uma base de dados, é feito uso de um recurso chamado de DataSet Tipado. O DataSet é introduzido, primeiramente, sem o uso de tabelas existentes um uma base de dados. Em seguida, são trabalhados recursos que buscam subsidiar uma agilidade no desenvolvimento, como a criação de um formulário que represente o CRUD (*Create, Retrieve, Update and Delete*), com operações de arrastar e soltar.

Capítulo 7 – Conhecendo a Language INtegrated Query e o Windows Presentation Foundation

Quando foi apresentado o ADO.NET, as consultas foram realizadas fazendo uso de instruções SQL, escritas como uma string. Depois, com a apresentação do Data Set Tipado, essas mesmas consultas foram transferidas para métodos, o que levou a uma aproximação à Orientação a Objetos. Este capítulo apresenta o *Language INtegrated Query* (LINQ) como ferramenta para realização dessas consultas, que podem ser realizadas em diversas fontes de dados, incluindo bases de dados. Com o LINQ, é possível fazer uso de objetos em sua sintaxe, permitindo uso de recursos do IDE, como o IntelliSense. Para aplicar o LINQ, também é feito uso do *Windows Presentation Foundation* (WPF), para implementação da interface gráfica com o usuário.

Capítulo 8 – Apresentando o Entity Framework como ferramenta para o Mapeamento Objeto Relacional

A persistência de dados, apresentada em capítulos anteriores a este, é realizada por meio do ADO.NET, quer seja fazendo uso de instruções SQL ou de componentes que encapsulam estas instruções (como o DataSet Tipado). Com o Entity Framework (EF), é possível trabalhar diretamente com objetos, não se preocupando em como a aplicação se conectará ao banco e como os dados serão capturados ou persistidos. O uso desse framework possibilita ao programador abstrair e trabalhar sempre com objetos. Para a aplicação dos exemplos, o capítulo faz uso também do WPF para interação com o usuário, que traz uma implementação do CRUD.

Capítulo 9 – Os estudos não param por aqui

Com este capítulo, concluímos este livro destacando todos os assuntos que vimos até aqui, junto de estímulos para que você continue sua jornada de aprendizagem e aplicação do C#.

Caso tenha alguma dúvida ou sugestão, procure a comunidade do livro para tirar dúvidas. Ela está disponível em https://groups.google.com/forum/#!forum/livro-windows-forms-casa-do-codigo. Lá podemos discutir mais sobre os temas tratados aqui. Você será muito bem-vindo!

Sobre o autor

Everton Coimbra de Araújo atua na área de treinamento e desenvolvimento.

É tecnólogo em processamento de dados pelo Centro de Ensino superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC e doutorado pela UNIOESTE em Engenharia Agrícola.

É professor da Universidade Tecnológica Federal do Paraná (UTFPR), campus Medianeira, onde leciona disciplinas no Curso de Ciência da Computação e em especializações.

Já ministrou aulas de Algoritmos, Técnicas de Programação, Estrutura de Dados, Linguagens de Programação, Orientação a Objetos, Análise de Sistemas, UML, Java para Web, Java EE, Banco de Dados e .NET.

Possui experiência na área de Ciência da Computação, com ênfase em Análise e Desenvolvimento de Sistemas, atuando principalmente nos seguintes temas: Desenvolvimento Web com Java e .NET e Persistência de Objetos.

O autor é palestrante em seminários de informática voltados para o meio acadêmico e empresarial.

Sumário

Versão: 18.9.2

Capítulo 1

As primeiras implementações: estrutura sequencial

Para as resoluções dos problemas apresentados neste livro, objetivou-se um padrão único de trabalho. Cada enunciado é acompanhado de uma análise sobre um problema e, nessa análise, é realizada uma explanação sobre ele, esclarecendo pontos importantes para sua compreensão e, consequentemente, auxiliando-o na resolução.

Faz parte também de cada enunciado a apresentação da Interface Gráfica com o Usuário desejada (*Graphical User Interface*, também conhecida como GUI) e detalhamentos sobre o funcionamento esperado. Toda implementação realizada em cada tópico é explicada, trazendo detalhes que possam merecer uma maior atenção por conta da lógica aplicada e dos recursos usados.

Sempre que for necessário utilizar recursos do IDE (*Integrated Develop-*

ment Environment) e acontecer de estes ainda não terem sido explicados, fiquem tranquilos! Eles serão apresentados de maneira detalhada. Isso também ocorrerá com os conceitos e as técnicas. Os problemas apresentados neste capítulo trabalham apenas a forma sequencial.

Uma **estrutura sequencial** é uma implementação (programa), na qual uma linha é executada após a outra, não ocorrendo desvios ou repetições. Praticamente todas as linguagens trabalham com essa estrutura de implementação. É a maneira mais simples de programação e é utilizada em conjunto com estruturas de seleção (condicional) e repetição, abordadas nos capítulos seguintes.

1.1 Troca de valores

Para realizar a solução do problema de troca de valores, crie uma aplicação, na qual sejam solicitados dois valores ao usuário em uma janela, não importando o tipo de dado neste momento. Em seguida, por meio da ação de um botão, faça os valores serem apresentados de maneira inversa ao usuário.

Como este livro trabalha aplicações Windows, os valores serão informados em controles que representam uma caixa de texto. Dessa maneira, essa troca do problema proposto refere-se ao processo de inversão entre os valores informados nessas caixas de texto (conhecidas e chamadas a partir de agora de `TextBox`). Por exemplo, o `TextBox` que recebe o primeiro valor deverá ter, em si, o valor informado e armazenado no segundo `TextBox`, que deverá ter o valor informado e armazenado no primeiro.

A figura 1.1 representa a janela responsável pela resolução para o problema proposto.

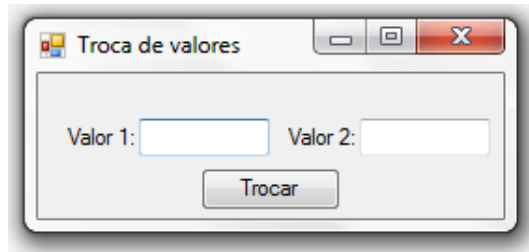



Fig. 1.1: Interface para o problema de troca de valores

Para a criação da aplicação no Visual Studio, representada pela figura 1.1, é preciso criar um projeto. Entretanto, para sua formação, é importante que ele faça parte de uma solução.

Uma **solução** (*solution*) é um projeto que pode ter outros diversos projetos dentro dele. Ela pode ser abstraída como uma aplicação, pois uma aplicação pode estar dividida em diversos projetos, como: um para a camada de apresentação; um para tratamento da lógica de negócio; outro para controlar as interações entre estas duas; e outro, ainda, para acessar um mecanismo de armazenamento de dados.

No Visual Studio, uma solução e seus projetos podem ser visualizados de maneira hierárquica na janela `Solution Explorer`. Para vê-la, acesse o menu `View → Solution Explorer` ou pressione a combinação de teclas `Ctrl + Alt + L`.

Para criar uma solução para o projeto, ainda nesse programa, selecione o menu `File → New Project`. Na janela que abre dentro da categoria, vá em `Installed → Templates → Other Project Types → Visual Studio Solutions` e selecione o template `Blank Solution`. No campo `Name`, digite **SolucaoCapitulo01** e, em `Location`, selecione a pasta onde ela será criada. Dentro da pasta selecionada será criada outra com o nome da solução. A figura 1.2 apresenta a janela para criação dessa solução, tendo destacadas as partes comentadas.

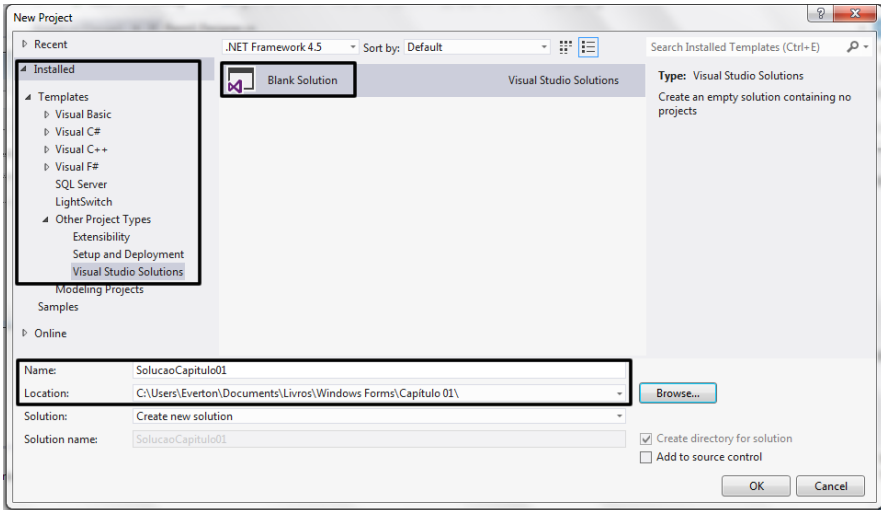



Fig. 1.2: Janela wizard para a criação de uma solução

Após a criação da *solution*, a janela `Solution Explorer` passa a exibi-la, como pode ser verificado na figura 1.3. Detalhes sobre as funcionalidades dessa janela serão apresentados e explicados quando forem necessários.

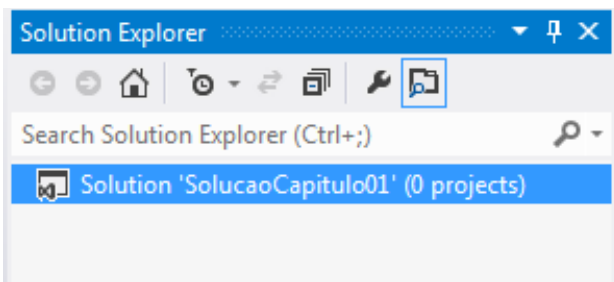



Fig. 1.3: Janela Solution Explorer

A solução criada anteriormente conterá todos os projetos que forem criados neste capítulo para efeito de organização. Para gerarmos o primeiro projeto, um dos caminhos é clicar com o botão direito do mouse em cima do nome da solução no `Solution Explorer` e, então, no menu de contexto,

escolher a opção `Add` → `New Project`. Na janela que abre, dentro da categoria `Installed` → `Visual C#` → `Windows`, selecione o *template* `Windows Forms Application`. No campo `Name`, digite **TrocaDeValores** e verifique se em `Location`, a pasta de destino é a da solução. Dentro da pasta selecionada será criada outra com o nome do projeto. A figura 1.4 apresenta a janela para sua criação, tendo destacadas as partes comentadas.

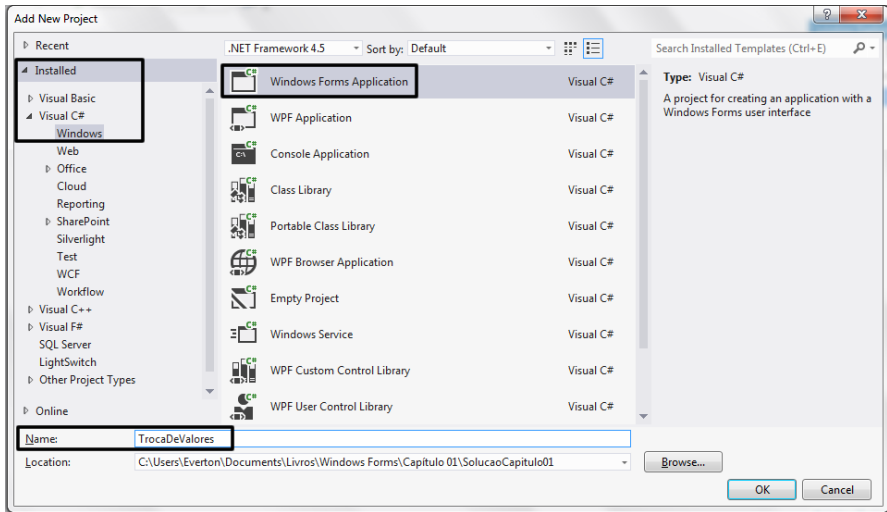



Fig. 1.4: Janela wizard para a criação de um projeto Windows Forms Application

Assim, a janela `Solution Explorer` passa a exibi-lo, como pode ser verificado na figura 1.5.

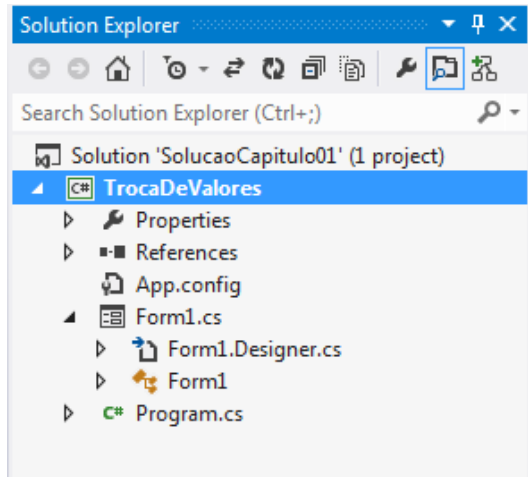



Fig. 1.5: Janela Solution Explorer com o projeto criado na solução

É possível identificar na hierarquia do projeto a existência de alguns nós (`Properties` e `References`) e alguns arquivos (`App.config`, `Form1.cs` e `Program.cs`). Ainda, como parte do arquivo `Form1.cs`, verifica-se outro, o `Form1.Designer.cs`. O detalhamento dos nós `Properties` e `References` e desses arquivos serão explicados conforme forem usados.

Após a criação de um projeto do tipo `Windows Forms Application`, o template utilizado cria, em conjunto com ele, duas classes: `Program` e `Form1`. Cada uma delas está implementada em arquivos com o mesmo nome. A classe `Form1` (que está no arquivo `Form1.cs`) estende `System.Windows.Forms.Form` e pode ser representada por uma janela no padrão Windows quando instanciada.

Objeto é qualquer estrutura modular que faz parte de algo. Ele pode ser algo físico (janela) ou um conceito (expressão matemática). Uma janela, por exemplo, é um objeto de uma casa, de um carro ou de um software com interface gráfica para o usuário (um formulário). Pense também em um livro como parte de uma biblioteca. Cada objeto possui `propriedades`, `comportamentos` e `métodos` que o identificam e diferenciam dentre outros semelhantes. **Instanciar** uma classe é o mesmo que obter um objeto de uma classe.

Classe é um conjunto ou uma categoria de objetos que têm atributos, propriedades e métodos (semelhantes às funções e aos procedimentos). É, na realidade, a implementação dos comportamentos que realizam algo específico já programado.

Para uma classe `Livro`, verifique que ela faz parte de uma classe mais geral, por exemplo, `Publicação`. A esta característica damos o nome de **Generalização**. Na classe `Publicação`, é possível detectar outras ainda mais específicas, como `Computação`, `Matemática`. A isso damos o nome de **Especialização**. **Extensão** é a especialização de uma classe. **Generalização** e **Especialização** são termos relacionados à **Herança** em Orientação a Objetos (OO).

A OO será utilizada de maneira gradativa neste livro. O capítulo 4 apresentará e aplicará essa metodologia de uma maneira mais aprofundada e específica.

O Visual Studio é um **Ambiente Integrado de Desenvolvimento** (IDE). Uma das ferramentas disponibilizadas por ele é a `Área de Desenho de Formulários`, destacada na figura a seguir.

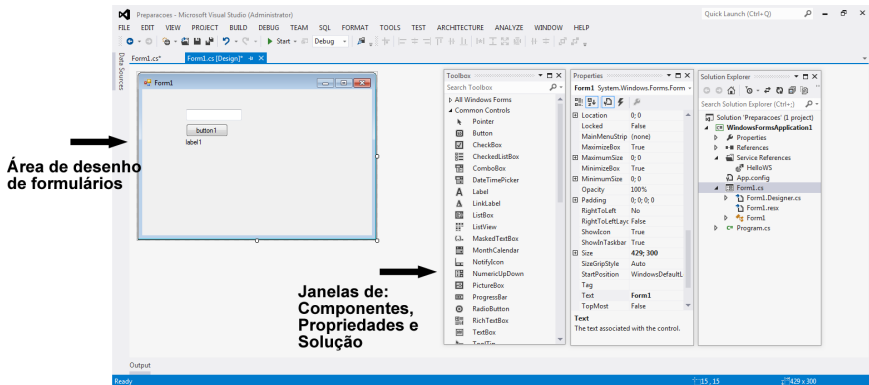



Fig. 1.6: IDE do Visual Studio com destaque para a Área de Desenho de Formulários

Um IDE é um ambiente composto por diversas ferramentas, dentre elas: compilador, editor de texto, ferramenta para depuração e alguns acessórios que trazem recursos a mais ao processo de desenvolvimento.

O desenho da janela, que representará a aplicação responsável pela resolução do problema da seção 1.1, se dará na classe criada em conjunto com o projeto, apresentada por padrão na área de desenho de formulários e identificada como `Form1`.

Essa atividade de desenho se dá por enquanto pelo processo de arrastar e soltar (*drag and drop*) controles para o formulário da janela, e de configurar as propriedades necessárias para cada um deles. Os controles possíveis de serem arrastados estão disponíveis na janela chamada `Toolbox`, que é exibida na figura a seguir.

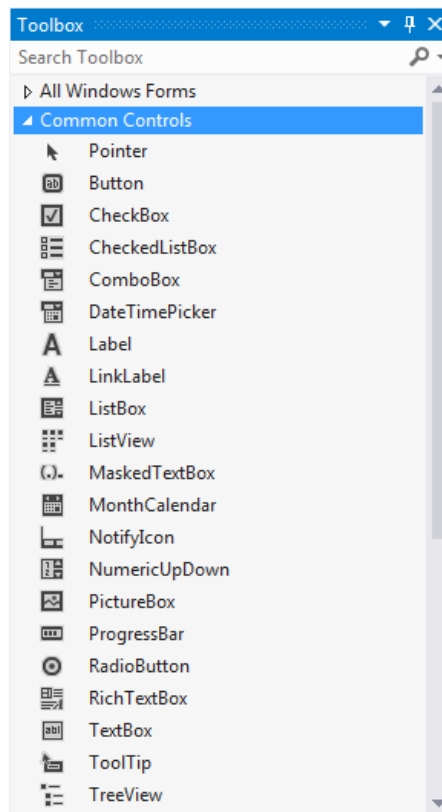



Fig. 1.7: Janela que representa a barra de ferramentas de controles possíveis para serem arrastados

Existem diversas categorias de controles nessa janela e elas serão apresentadas de acordo com a necessidade de uso. Neste exemplo, os controles que usaremos fazem parte da categoria `Common Controls`.

Para a aplicação exibida na figura 1.1, será preciso arrastar três componentes para o formulário da área de desenho: um `Label`, dois `TextBox` e um `Button`. A princípio, procure puxar os componentes e organizá-los da maneira apresentada. O Visual Studio oferece as "linhas guias" que orientam o posicionamento vertical e horizontal. Isso pode ser verificado pela figura 1.8, que tem um `Label` já adicionado e mostra um `TextBox` sendo arrastado.

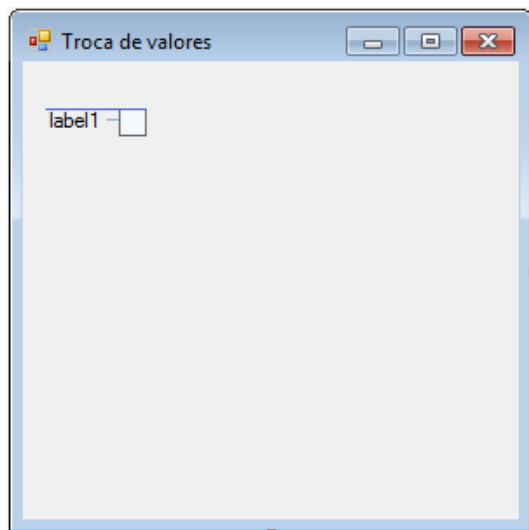



Fig. 1.8: Janela apresentando linhas guias no momento de arrastar componentes para o formulário

Cada controle inserido no formulário, assim como o próprio formulário, possuem propriedades que precisam ser configuradas para que o resultado seja o esperado. Para configurar as propriedades de um controle, é preciso que ele esteja selecionado, bastando apenas clicar nele. Feito isso, é necessário acessar a Janela de Propriedades (`Properties Window`), exibida na figura 1.9.

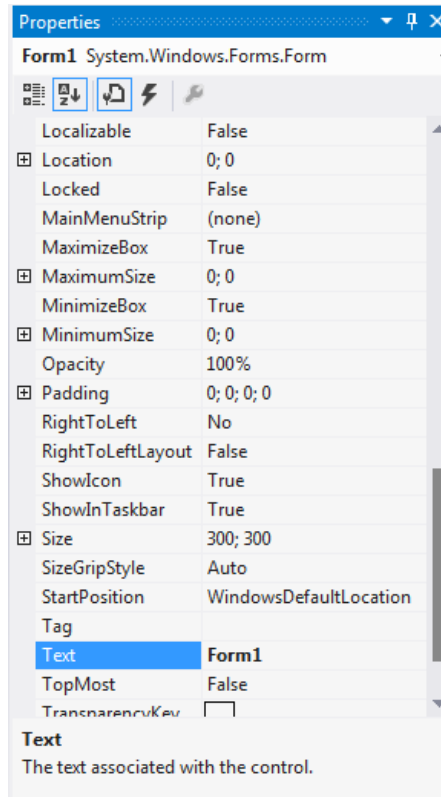



Fig. 1.9: Janela de configuração de propriedades de controles (Properties Window)

Um formulário é um objeto da classe `System.Windows.Forms.Form` que representa uma janela ou uma caixa de diálogo, que podem ser vistas como interface com o usuário em aplicações. A classe `Form` pode ser usada para criar janelas padrões, de ferramentas, sem bordas e flutuantes. Estas podem ser `Modal`, como uma janela de diálogo.

Utilizando as propriedades disponíveis na classe `Form`, é possível determinar a aparência, tamanho e cor, além de características de gerenciamento de uma janela criada.

> A documentação sobre a classe `Form` pode ser verificada na MSDN (*Microsoft Developer Network*) em http://bit.ly/1KDiOM5. Sempre que houver alguma dúvida, procure acessar a MSDN, pois ela é a referência oficial para os recursos utilizados neste livro.

Para acessar a `Properties Window`, vá ao menu `View` → `Properties Window` ou pressione a tecla `F4`.

Uma característica da janela que vamos criar é não permitir que o usuário altere seu tamanho; algo possível em algumas ao arrastar de bordas. Este processo, sem ter no formulário uma configuração adequada de suas propriedades e de seus controles, fará com que a janela seja exibida de uma maneira visual errada.

As propriedades do formulário e dos controles podem ser atribuídas por meio de codificação, além da `Properties Window`, quando houver necessidade. Altere o valor das propriedades para esse formulário conforme apresentado na sequência.

Propriedades para o formulário

- **Propriedade:** FormBorderStyle
- **Valor para atribuir:** FixedSingle

A propriedade **FormBorderStyle** é responsável pelo estilo da borda da janela que o formulário representará. Ele pode ter os seguintes estilos de borda: `FixedSingle`, `Fixed3D`, `FixedDialog`, `Sizable`, `FixedToolWindow` e `SizableToolWindow`. No momento, serão detalhadas apenas `Sizable` e `FixedSingle`.

A `Sizable`, que é o estilo padrão, permite que o usuário altere o tamanho da janela durante sua execução. Já a `FixedSingle` não permite essa mudança, ou seja, a janela terá as dimensões atribuídas em tempo de desenho.

- **Propriedade:** Text

- **Valor para atribuir:** Troca de valores

A propriedade `Text` é responsável pela identificação de um objeto, como o nome dado a uma variável. Não há necessidade de todo objeto ter um nome específico atribuído, pois, quando um controle é arrastado, o objeto é criado e um nome é automaticamente atribuído. Entretanto, caso ele seja usado na codificação de sua aplicação, é altamente recomendado que o nomeie. Esse nome pode seguir uma convenção, na qual os primeiros três dígitos caracterizam o tipo do objeto – como **frm** para formulário – e o restante uma descrição que associe facilmente ao seu conteúdo. Todo controle possui essa propriedade.

- **Propriedade:** Name
- **Valor para atribuir:** frmTrocaDeValores

A propriedade `Name` – que aparece como `(Name)` na janela de propriedades – é responsável pela identificação de um objeto, como o nome dado a uma variável. Como dito anteriormente, não é necessário nomear o objeto, porém, é recomendado, para caso ele seja utilizado em algum código.

- **Propriedade:** Size (*Width* e *Height*)
- **Valor para atribuir:** 265 para Width e 115 para Height

A propriedade `Size` é responsável pela definição do tamanho do formulário. Ela é subdividida em duas subpropriedades: `Width` para comprimento e `Height` para altura. Também é uma propriedade comum em controles que são ditos "visuais".

Os controles podem ser divididos em **visuais** e **não visuais**. Os visuais podem ser exibidos em formulários e os não visuais são disponibilizados para o trabalho por meio de código, ou como um auxiliar a outros controles.

Em relação aos `Label`s arrastados para o formulário, o valor exibido é **LabelX**, onde **X** é o valor incrementado de acordo com a quantidade de `Label`s arrastados. Esse padrão de nomeação de controles é o mesmo para

todos os tipos. É lógico que é preciso deixar um texto que oriente o usuário, como exibido na figura 1.1. Desta maneira, as propriedades que precisam ser configuradas para os `Labels` são apresentadas na sequência. Propriedades já descritas terão apenas seus nomes e valores apresentados para serem atribuídos.

Um `Label` é um objeto da classe `System.Windows.Forms.Label` que representa um texto para orientar o usuário normalmente acompanhado de uma caixa de texto (`TextBox`). Assim como o `Form`, ele e a maioria dos controles visuais permitem configurações relacionadas à sua aparência.

Propriedades para o primeiro Label

- **Propriedade:** Location (X e Y)
- **Valor para atribuir:** 13 para X e 27 para Y

A propriedade `Location` é responsável pela definição da **posição** do controle em seu contêiner, que, neste caso, é o formulário. Ela é composta por duas subpropriedades: **X** para a posição horizontal (esquerda do *container*) e **Y** para a vertical (seu topo).

- **Propriedade:** Text
- **Valor para atribuir:** Valor 1:

Propriedades para o segundo Label

- **Propriedade:** Location (X e Y)
- **Valor para atribuir:** 131 para X e 27 para Y
- **Propriedade:** Text
- **Valor para atribuir:** Valor 2:

As propriedades que precisam ter seus valores configurados para os dois `TextBox`s são exibidas na sequência. A propriedade `Text` existe também para o `TextBox` e tem a mesma finalidade. Entretanto, como o objetivo do `TextBox` é permitir a interação com o usuário para que ele informe valores que possam ser trabalhados pela aplicação, dificilmente essa propriedade tem um valor atribuído em tempo de desenho.

Um `TextBox` é um objeto da classe `System.Windows.Forms.TextBox`, que representa uma caixa de texto, na qual o usuário poderá digitar valores que servirão como dados de entrada para a aplicação. O valor informado é atribuído na propriedade `Text` e, assim como todas as propriedades, pode ser acessada por meio de código.

Propriedades para o primeiro TextBox

- **Propriedade:** Name
- **Valor para atribuir:** txtPrimeiroValor

- **Propriedade:** Location (X e Y)
- **Valor para atribuir:** 54 para X e 24 para Y

- **Propriedade:** Size (Width e Height)
- **Valor para atribuir:** 70 para Width e 20 para Height

Propriedades para o segundo TextBox

- **Propriedade:** Name
- **Valor para atribuir:** txtSegundoValor

- **Propriedade:** Location (X e Y)
- **Valor para atribuir:** 172 para X e 24 para Y

- **Propriedade:** Size (Width e Height)
- **Valor para atribuir:** 70 para Width e 20 para Height

Propriedades para o Button

Para finalizar este projeto, é preciso configurar o controle `Button`. Essa configuração é apresentada a seguir.

Um `Button` é um objeto da classe `System.Windows.Forms.Button` que representa um botão de ação, no qual algum tipo de processamento é disparado quando o usuário o clica.

- **Propriedade:** Image
- **Valor para atribuir:** À escolha do leitor

Detalhes sobre a propriedade `Image` são apresentados na seção 1.2.

- **Propriedade:** Text
- **Valor para atribuir:** Valor 1:

- **Propriedade:** ImageAlign
- **Valor para atribuir:** Middle Left

A propriedade `ImageAlign` é responsável pela definição da **posição da imagem** no controle em que ela será inserida. Na escolha do valor no `Properties Window`, as opções aparecem em forma de desenho.

- **Propriedade:** Location (X e Y)
- **Valor para atribuir:** 87 para X e 50 para Y

- **Propriedade:** Size (Width e Height)
- **Valor para atribuir:** 75 para Width e 23 para Height

- **Propriedade:** TextAlign

- **Valor para atribuir:** Middle right

A propriedade `TextAlign` é responsável pela definição da **posição do texto** no controle. Na escolha do valor no `Properties Window`, as opções aparecem em forma de desenho, tal como a propriedade `ImageAlign`.

1.2 Inserindo imagens em controles

Alguns controles visuais, como no caso do `Button`, permitem que imagens sejam exibidas por eles. Desta maneira, ao acessar a propriedade `Image` desses componentes, uma janela é exibida para a importação ou referência da imagem que será usada.

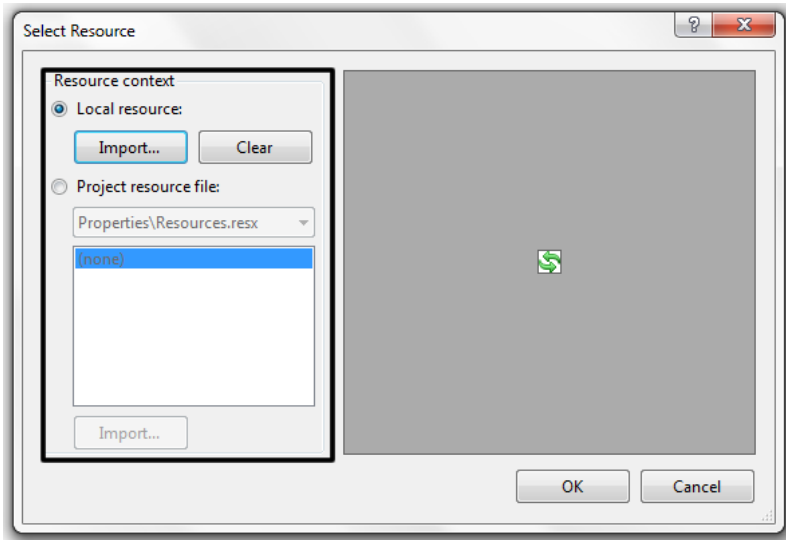



Fig. 1.10: Janela para seleção e importação de imagens

Na figura 1.10, existem duas maneiras para importação de uma imagem. A primeira, `Local resource`, importa uma que esteja armazenada em seu disco para o controle em questão. Dessa maneira, se a imagem introduzida for usada mais de uma vez, será preciso importá-la a cada necessidade de uso. Para facilitar esse seu uso múltiplo, existe a segunda opção: `Project`

`resource file`. Uma imagem importada por essa opção a trará para um arquivo chamado `Resources`, deixando-a disponível para reúso em seu projeto.

> **Dica:** O site http://www.iconfinder.com/ oferece uma excelente ferramenta para obtenção de imagens.

A interface com o usuário está concluída. Todos os controles inseridos em um formulário causam a escrita de um código referente a eles na classe que os representa, neste caso `Form1`. Para verificar essa codificação criada, no `Solution Explorer` expanda o nome do formulário criado e dê um duplo clique no arquivo `Form1.Designer.cs`.

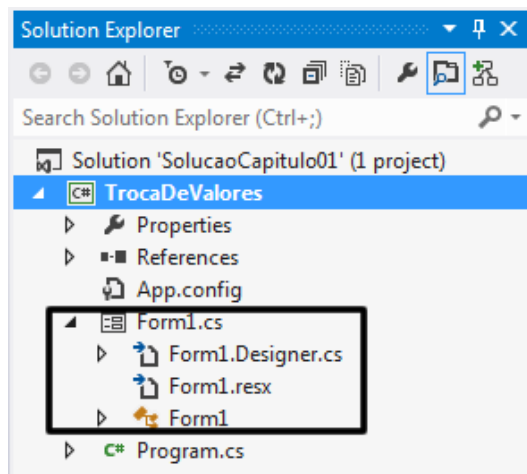



Fig. 1.11: Solution Explorer exibindo subarquivos para formulário

Ao abrir o arquivo no editor de códigos, é possível verificar a implementação da classe `Form1`, ilustrada na figura 1.12. Na linha 1, verifica-se a declaração do `Namespace` que conterá a classe; a linha 2 declara a classe em si; e as declarações entre as linhas 102 e 106 referem-se a todos os controles arrastados para o formulário.

```
1   namespace TrocaDeValores {
2       partial class Form1 {
3           /// <summary>
4           /// Required designer variable.
5           /// </summary>
6           private System.ComponentModel.IContainer components = null;
7
8           /// <summary>
9           /// Clean up any resources being used.
10          /// </summary>
11          /// <param name="disposing">true if managed resources should
12          /// be disposed; otherwise, false.</param>
13          protected override void Dispose(bool disposing)
14          {
15              if (disposing && (components != null))
16              {
17                  components.Dispose();
18              }
19              base.Dispose(disposing);
20          }
21
22          Windows Form Designer generated code
101
102         private System.Windows.Forms.Label label1;
103         private System.Windows.Forms.TextBox txtPrimeiroValor;
104         private System.Windows.Forms.Label label2;
105         private System.Windows.Forms.TextBox textBox2;
106         private System.Windows.Forms.Button button1;
107     }
108 }
```

Fig. 1.12: Parte da classe Form1 que contém código relacionado aos componentes inseridos no formulário

Um `Namespace` é um recurso que permite organizar logicamente as classes, criando uma espécie de estrutura em árvore, como se fossem diretórios ou pastas. Dessa maneira, toda classe possui seu nome de maneira qualificada, ou seja, ele completo. No caso da classe `Form1`, seu nome qualificado é `TrocaDeValores.Form1`.

O C# possui um recurso muito interessante que permite que uma mesma classe possua implementação em diversos arquivos. Isso é possível por meio da definição de classes com o uso da palavra reservada `partial`.

Ao executarmos a aplicação, ela funcionará perfeitamente no sentido de aceitar que o usuário digite os dois valores, mas nada mais que isso, mesmo

ao pressionar o botão `Trocar Valores`. Para rodá-la, acesse o menu `Debug → Start Without Debugging` ou pressione a combinação de teclas `Ctrl + F5`. Ainda na barra de tarefas, é possível clicar no botão `Start`, selecionando `Release` no *combobox* que exibe `Debug`. Esta é a maneira mais leve; entretanto, caso queira (ou necessite), é possível também executar com as opções oferecidas para `Debug`.

Debug (depuração) é o processo de investigar uma aplicação, buscando corrigir erros e melhorar performance. Para isso, é necessário ferramentas específicas, que já vêm instaladas e integradas no Visual Studio.

Para que a aplicação apresente o comportamento desejado no momento em que o usuário clicar no botão, deve-se implementar esta funcionalidade. No entanto, toda interface de usuário criada é orientada a eventos. Desta maneira, é preciso identificar um que ocorra no momento desejado. Nos exemplos propostos, esse momento dever ser *quando* o usuário *clicar* no botão.

É claro que é difícil adivinhar quais eventos estão disponíveis. Para isso, o Visual Studio, por meio da janela `Properties Window`, exibe todos os eventos possíveis de serem capturados para cada controle disponível no formulário. Com o botão selecionado, clique em `Events`, ilustrado com o ícone de um relâmpago no topo da `Properties Window`, e os eventos disponíveis serão exibidos.

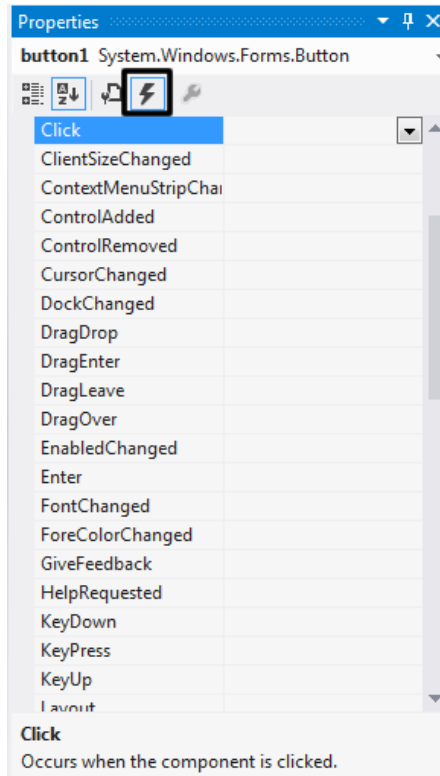



Fig. 1.13: Properties Window exibindo os eventos para um controle

O evento `Click`, que aparece como primeiro evento disponível na relação exibida na figura 1.13, é o responsável por capturar o momento em que o usuário clica no botão. Para implementar esse comportamento desejado, realize um duplo clique na área em branco, ao lado do nome dele. Isso causará a abertura do editor de código já com o método que recebe a delegação desse evento criado (figura 1.14).

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10
11 namespace TrocaDeValores
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void button1_Click(object sender, EventArgs e)
21         {
22
23         }
24     }
25 }
```

Fig. 1.14: Estrutura do método que recebe a delegação para tratamento do evento Click

Todo controle possui um **evento padrão** e, no caso do `Button`, esse evento é o `Click`. Essa informação é útil caso deseja-se acessar diretamente o código implementado para ele, no editor visual do formulário. Para isso, basta realizar um duplo clique no próprio controle.

No código da figura 1.14, nas linhas 11 e 13, é possível constatar a declaração do `Namespace` da classe, da mesma maneira que foram apresentados os códigos gerados pelos controles arrastados para o formulário. É importante saber, como explicado sobre `partial class`, que a classe desse arquivo é a mesma da anteriormente apresentada. O código entre as linhas 15 e 18 referem-se ao construtor da classe.

Métodos **construtores** são responsáveis pela inicialização de um objeto,

preparando-o para ser usado.

O método `button1_Click()` – que no prefixo de seu nome tem o nome do controle e o sufixo com o nome do evento – recebe a delegação de atender ao evento quando ele ocorrer. Esse é o padrão para todos os nomes de métodos que respondem a eventos e que sejam criados de maneira automática pelo Visual Studio. A delegação pode ser visualizada no arquivo que tem todos os códigos gerados de maneira automática, apresentado anteriormente na figura 1.12. Veja esse código na figura a seguir.

```
63            // textBox2
64            //
65            this.textBox2.Location = new System.Drawing.Point(172, 24);
66            this.textBox2.Name = "textBox2";
67            this.textBox2.Size = new System.Drawing.Size(70, 20);
68            this.textBox2.TabIndex = 3;
69            //
70            // button1
71            //
72            this.button1.Image = ((System.Drawing.Image)(resources.GetObject("button1.Image")));
73            this.button1.ImageAlign = System.Drawing.ContentAlignment.MiddleLeft;
74            this.button1.Location = new System.Drawing.Point(87, 50);
75            this.button1.Name = "button1";
76            this.button1.Size = new System.Drawing.Size(75, 23);
77            this.button1.TabIndex = 4;
78            this.button1.Text = "Trocar valores";
79            this.button1.TextAlign = System.Drawing.ContentAlignment.MiddleRight;
80            this.button1.UseVisualStyleBackColor = true;
81            this.button1.Click += new System.EventHandler(this.button1_Click);
```

Fig. 1.15: Código de configurações criado automaticamente para propriedades e eventos (arquivo Form1.Designer.cs)

Note que parte da interação com o usuário, que é permitir que sejam informados os valores desejados, está concluída. A troca desses valores ocorrerá quando o botão `Trocar Valores` for pressionado. O código a ser implementado é apresentado na sequência.

Resolução 1.1: evento Click para o Button de troca de valores

```
private void button1_Click(object sender, EventArgs e) {
    string auxiliar;
    auxiliar = txtPrimeiroValor.Text;
    txtPrimeiroValor.Text = txtSegundoValor.Text;
    txtSegundoValor.Text = auxiliar;
```

```
    MessageBox.Show("Troca de valores concluída",
        "Informação", MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

O evento `Click` é capturado no momento em que o `Button` – ou outro controle que possua esse evento – é pressionado: ou pelo botão esquerdo do mouse, ou pelo foco e a tecla `Enter` (ou `Barra de Espaço`) em conjunto. A maioria dos controles visuais que interagem com o usuário por meio do mouse tem esse mesmo comportamento. Dessa maneira, em resoluções futuras, esse evento não será mais descrito. Essas definições são válidas para todos que trazem esse evento.

O primeiro ponto a ser tratado no código é a declaração da variável **auxiliar**, que tem seu tipo definido como `string`. Isso deve-se ao fato de a propriedade `Text` do componente `TextBox`, que receberá o valor informado pelo usuário, também ser definida assim. Veja que a variável `auxiliar` do código anterior recebe o valor existente no primeiro `TextBox` e este, por sua vez, recebe o valor existente no segundo. Porém, este segundo recebeu o valor que estava no primeiro anteriormente e que agora está na variável auxiliar.

É possível que em alguns exemplos ou livros que você possa ler, use `String` e `string`. Ambas definem um objeto como string; entretanto o uso do primeiro caso refere-se ao tipo de dado definido pelo .NET, como base de implementação para outras linguagens, sendo essa a classe do `namespace System.String`. Já `string` é o tipo de dado da linguagem C#.

O C# utiliza um padrão para que se trabalhe com a propriedade de um controle, seja para receber um valor como está sendo feito nesse nosso código (atribuição), ou para obter o valor atribuído a uma propriedade (retorno). Dessa maneira, a sintaxe da instrução sempre deve referenciar o `NomeDoControle` e a `propriedade` desejados e separados por um ponto.

Após a execução das instruções, ao rodarmos a aplicação, informarmos os valores e clicarmos no botão, uma mensagem de término é exibida, como pode ser visto na figura a seguir.

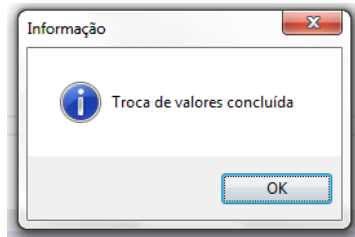



Fig. 1.16: Mensagem de confirmação da troca de valores

O método estático `Show()` da classe `MessageBox` foi utilizado para informar ao usuário o término do processamento referente à troca de valores entre os controles. Diversas classes e métodos nativos do C# serão usados e, a medida que o são, terão suas descrições apresentadas.

A classe `System.Windows.Forms.MessageBox` é responsável por fornecer métodos estáticos para a geração de janelas informativas e de interação com o usuário.

Métodos normalmente estão ligados aos objetos, sendo eles exclusivos de cada instância. Entretanto, existem situações em que um pode pertencer à classe, podendo ser utilizado para qualquer objeto dela. Eles são conhecidos como **métodos estáticos**, pois não precisam de um objeto e podem ser acessados diretamente da classe. É preciso ter ciência de que qualquer variável declarada nessa classe será acessível para todos os seus objetos, e o valor dessa variável será sempre o mesmo para todos.

A assinatura usada no método `Show()` em nosso exemplo é a `public static DialogResult Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon)`. Nela são enviados: o texto a ser exibido ao usuário, o título para a janela, os botões disponibilizados, e o ícone a ser exibido na barra de títulos. O retorno refere-se a qual botão foi pressionado pelo usuário.

O método estático `System.Windows.Forms.MessageBox.Show()` tem como responsabilidade a exibição de mensagens para o usuário, permitindo uma interação. Ele possui diversas assinaturas que merecem um estudo aprofundado. A documentação oferecida pela Microsoft é rica, bem escrita e com diversos exemplos.

Um método tem por objetivo realizar algum tipo de processamento. É comum que o problema a ser resolvido por ele possua um conjunto variável de parâmetros. Quando isso ocorre, alguns tendem a desenvolver diversos métodos, um para cada quantidade e para tipos de parâmetros recebidos. Entretanto, quando todos resolvem o mesmo problema, é correto que ele tenha o mesmo nome, variando apenas na quantidade e tipos de parâmetros que recebe. A isso damos o nome de **sobreposição**, e a cada conjunto de parâmetros diferentes para um mesmo método chamamos de **assinatura**. O tipo de retorno não faz parte da assinatura quando se trata de sobreposição.

Ao executarmos a aplicação, o que notamos é o surgimento do formulário na janela. Entretanto, antes de ele ser exibido, é preciso instanciar sua classe. Verifique no `Solution Explorer` a existência de um arquivo chamado `Program.cs`. Ele refere-se a uma classe de mesmo nome, com um método estático especial, `Main()`.

Veja na figura 1.17 que a terceira linha desse método invoca o estático `Application.Run()`, enviando como argumento a instância da classe `Form1`. Ao instanciá-la, seu construtor é executado, invocando o `InitializeComponent()`, declarado na mesma classe, porém no arquivo `Form1.Designer.cs`. O construtor está definido no arquivo `Form1.cs`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace TrocaDeValores
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Fig. 1.17: Método Main() invocando o formulário criado

O método `Main()`, diferente de qualquer outro, é conhecido como *ponto de entrada* para uma aplicação C# ou Windows. Quando uma aplicação é iniciada, ele é o primeiro a ser chamado.

1.3 Obtendo o resto de uma divisão

Agora, para trabalharmos um novo exemplo e novos recursos, será implementando um programa que permite o usuário informar dois números e que, ao pressionar um botão, faça a aplicação retornar o resto da divisão do primeiro número pelo segundo.

Normalmente, em uma operação de divisão, o desejado é seu quociente. Entretanto, nesse problema proposto, o que nos interessa é o seu *resto*; ou seja, quanto faltou para que a divisão fosse exata. Para isso, devemos considerar que o quociente é obrigatoriamente um número inteiro. O C# oferece um operador de resto, conhecido como módulo, e que, na linguagem, é represen-

tado pelo símbolo de percentagem (%).

A figura 1.18 representa a interface desejada que será implementada para o problema proposto nesta seção.

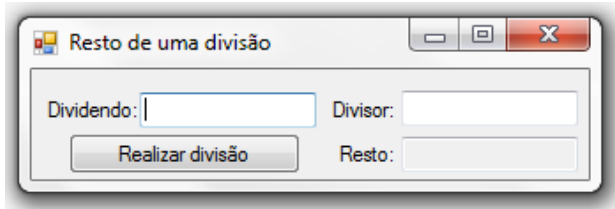



Fig. 1.18: Interface para o problema de obtenção do resto de uma divisão

Siga os mesmos passos da criação do projeto anterior para a criação desse segundo, dando ao projeto o nome `RestoDaDivisao`. No anterior, utilizamos o formulário criado pelo template, o `Form1`. Porém, o ideal é que cada formulário tenha um nome que o represente. Assim, elimine esse formulário do projeto, pressionando o botão direito e escolhendo a opção de remoção, ou selecionando o formulário e pressionando a tecla `delete` de seu teclado.

Para criar um formulário no projeto, clique com o botão direito do mouse sobre seu nome e selecione `Add` → `Windows Form`. Confirme a linguagem C# e veja se o template `Windows Forms` está selecionado, como demonstra a figura a seguir. Nomeie o formulário como `FormRestoDeDivisao.cs`.

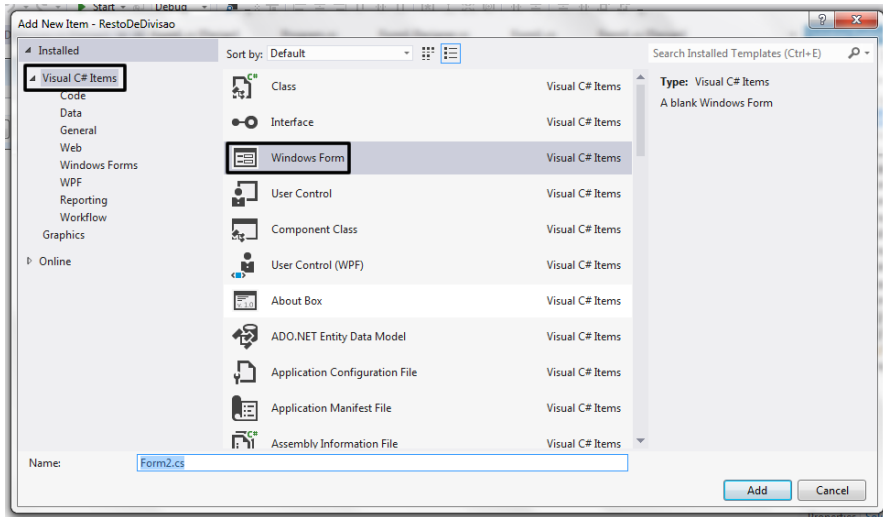



Fig. 1.19: Janela para seleção de template na criação de um Windows Forms

Após a criação do formulário, arraste os controles para que fiquem semelhantes ao apresentado na figura 1.18. São os mesmos vistos no exemplo anterior.

Em aplicativos que fazem uso de janelas, cada uma delas possui um conjunto padrão de ícones, os quais permitem que ela seja minimizada, maximizada, fechada e movida de lugar. Entretanto, existem janelas que não devem ser minimizadas ou maximizadas, o que leva a não necessidade desses botões. Neste exemplo, essa característica é trabalhada.

Outro atributo para essa aplicação é a existência de um controle de entrada (`TextBox`) que não deverá permitir essa interação, ou seja, apenas exiba dados. Na sequência, são apresentadas as descrições para as propriedades de formulário ainda não vistas e os valores das já conhecidas.

Propriedades para o formulário

- **Propriedade:** ControlBox
- **Valor para atribuir:** True

A propriedade `ControlBox` é responsável por definir se o ícone da janela conhecido como `System Menu`, exibido no lado esquerdo da barra de título, será mostrado ou não. Atribuindo essa propriedade para `False`, todos os botões da barra de título deixam de ser exibidos. O fechamento da janela pode ocorrer pelo pressionamento conjunto das teclas `Alt + F4` ou por algum controle que implemente essa função.

- **Propriedade:** FormBorderStyle
- **Valor para atribuir:** FixedSingle

- **Propriedade:** MaximizeBox
- **Valor para atribuir:** False

A propriedade `MaximizeBox` é responsável por definir se o botão de maximização de uma janela (`Maximize`), exibido em conjunto com outros dois botões (`Minimize` e `Close`) do lado direito da barra de título, será mostrado ou não.

- **Propriedade:** MinimizeBox
- **Valor para atribuir:** True

A propriedade `MinimizeBox` é responsável por definir se o botão de minimização de uma janela (`Minimize`), exibido em conjunto com outros dois botões (`Maximize` e `Close`) do lado direito da barra de título, será mostrado ou não.

- **Propriedade:** Name
- **Valor para atribuir:** frmRestoDeDivisao

- **Propriedade:** Size (Width e Height)
- **Valor para atribuir:** 331 para Width e 100 para Height

- **Propriedade:** Text

- **Valor para atribuir:** Resto de uma divisão

No formulário criado para a aplicação, insira três `Labels`, três `TextBoxs` e um `Button`. Nos `Labels`, altere apenas a propriedade `Text`, de acordo com o apresentado na figura 1.18. Use-a também para posicionar todos os controles no formulário.

Nos `TextBoxs`, nomeie-os de acordo com seu objetivo. Em caso de dúvidas, verifique as observações relativas a elas no exemplo anterior. O `TextBox` que terá a responsabilidade de exibir o resto da divisão terá suas propriedades específicas apresentadas na sequência.

Caso seja preciso atribuir um mesmo valor a mais de um controle, é possível selecioná-los e fazer isso de uma única vez. Para isso, selecione o primeiro controle, clicando sobre ele com o botão esquerdo do mouse. Para os demais, a seleção precisará ser com a tecla `Ctrl` pressionada. Após ter todos os controles escolhidos, defina a propriedade desejada e atribua a ela o valor.

Propriedades para o TextBox

- **Propriedade:** Name
- **Valor para atribuir:** txtRestoDaDivisao
- **Propriedade:** ReadOnly
- **Valor para atribuir:** False

A propriedade `ReadOnly` é responsável por definir se o controle deverá ou não receber o foco durante a navegação entre os controles por meio do pressionamento da tecla `Tab`. Caso o valor seja `False` e o usuário clique nele, ele receberá o foco.

Todo controle visual que permite a interação com o usuário possui duas propriedades responsáveis pela navegação do cursor, quando utilizada a tecla `Tab`. Elas são `TabIndex`, que mantém a ordem do controle dentro de seu contêiner; e `TabStop`, que informa se o controle receberá ou não o foco

quando a tecla Tab for pressionada. Caso deseje-se que ele não receba realmente o foco e que também não permita nenhuma interação, servindo apenas para exibição de dados, pode-se optar em atribuir False à propriedade Enabled.

Resolução 1.2: evento Click para o Button de cálculo do resto da divisão

```
private void button1_Click(object sender, EventArgs e){
    int dividendo = Convert.ToInt32(txtDividendo.Text);
    int divisor = Convert.ToInt32(txtDivisor.Text);
    int resto = dividendo % divisor;
    txtResto.Text = resto.ToString();
}
```

As duas primeiras linhas do corpo do método realizam um processo de conversão, e o resultado desse procedimento é atribuído a uma nova variável que existirá apenas dentro do método. A terceira linha realiza a operação de módulo por meio do operador % atribuindo o resto da divisão à variável resto.

Normalmente, toda entrada de dados por meio de interação de controles visuais tem o valor informado armazenado em uma propriedade do tipo string. Esses valores, como no exemplo apresentado, podem ser necessários em alguma operação que utilize um tipo de dado diferente, o que leva à necessidade de **converter** o dado para o tipo desejado. A classe Converter oferece vários métodos estáticos, que recebem um valor em um determinado tipo e o converte, retornando o dado no tipo desejado. No exemplo, o valor enviado para o método ToInt32() é uma string e será retornado por ele um Int32 (tipo .NET), que refere-se ao int no C#. Busque investigar essa classe e seus métodos na documentação do C#.

O método ToString() existe em toda classe, já que ele é definido na classe Object, possíveis todas as classes estendem-se dela. Sua funcionalidade é retornar uma string do objeto em que o método é invocado. O ToString() pode ser sobrescrito em qualquer classe, caso seja necessário.

Toda variável, classe, campo ou método tem uma característica em relação à sua visibilidade, ou **escopo**. Estes termos referem-se a *quem* pode aces-

sar o recurso e qual o tempo de vida dele. No método apresentado, algumas variáveis foram declaradas e, em seguida, inicializadas. A declaração de uma variável ou de um recurso, por si só, define seu escopo. Como, no exemplo, foram declaradas dentro de um método, o tempo de vida delas é o tempo de vida do método, e a visibilidade é apenas dentro dele.

É possível implementar todo o código apresentado em quatro linhas, em apenas uma e isso é uma prática comum. Porém, em alguns casos, isso pode torná-lo inelegível. Veja:

```
txtResto.Text = (Convert.ToInt32(txtDividendo.Text) %
    Convert.ToInt32(txtDivisor.Text)).ToString();
}
```

Terminada toda a implementação e configuração do formulário, é preciso testá-lo. Como essa aplicação faz parte de uma solução que já possui um projeto, é preciso informar ao Visual Studio qual projeto deve ser executado por padrão. Para isso, clique com o botão direito do mouse sobre seu nome, selecione a opção `Set as Startup Project` e execute-o.

Por meio da figura 1.20, é possível verificar a janela após a informação dos valores e o pressionamento no botão, que realiza a solução do problema proposto.

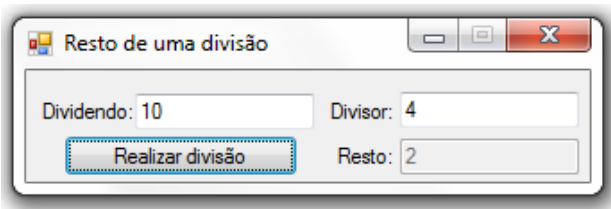



Fig. 1.20: Janela com a solução implementada

Quando trabalhamos com vários projetos em uma solução, para identificar visualmente qual está atualmente em edição, basta verificar qual deles tem o nome em **negrito**.

A implementação realizada não trabalhou a hipótese de o usuário digitar no divisor o valor 0 (zero). Essa verificação depende de recursos ainda não trabalhados.

1.4 Registrando o gasto em um restaurante

Para finalizar os exemplos deste capítulo, implemente um programa que permita a digitação de um valor referente a um consumo qualquer em um restaurante. Após o usuário informar esse valor, o sistema precisa calcular a taxa de serviço a ser repassada ao garçom: 10% do valor total da conta.

Algumas características visuais serão trabalhadas na implementação desse problema, como estilo visual dos `Labels` e `TextBoxs`, em relação às cores e formatação, e o posicionamento da janela quando a aplicação for executada. O problema apresentado pelo enunciado é simples e, com os recursos já trabalhados, é possível realizar a implementação.

A figura 1.21 representa a interface desejada, que será implementada para o problema proposto nesta seção.

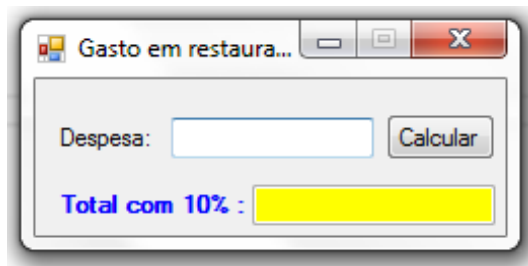



Fig. 1.21: Interface para o problema de obtenção do gasto total em um restaurante

Crie um novo projeto de nome `GastoEmRestaurante` na solução já existente. Nele, elimine o formulário criado pelo template e crie um chamado `FormGastoEmRestaurante`. Não existem novos controles para serem arrastados; são novamente `Labels`, `TextBoxs` e `Button`. Sendo assim, arraste-os e posicione-os conforme a figura 1.21. As novas propriedades trabalhadas nesse formulário são apresentadas na sequência.

Propriedades para o formulário

- **Propriedade:** StartPosition

- **Valor para atribuir:** CenterScreen

A propriedade `StartPosition` é responsável por definir a posição inicial da janela quando for exibida na primeira vez. Além do valor a ser atribuído, existem as opções: `Manual`, `WindowsDefaultLocation`, `WindowsDefaultBounds` e `CenterParent`.

Em relação aos controles, apenas as propriedades relacionadas a suas aparências ainda não foram apresentadas. Isso será feito a seguir. As demais, necessárias para o ajuste de posição e tamanho usados nos controles, já são conhecidas. Dessa maneira, arrume os controles para assemelharem-se ao layout apresentado pela figura 1.21.

Propriedades para o Label do campo Total

- **Propriedade:** Font (Bold)
- **Valor para atribuir:** True para Bold

A propriedade `Font` é composta por diversas subpropriedades que permitem a configuração de um estilo visual para o texto que o `Label` exibe. Qualquer alteração poderá ser verificada imediatamente no controle. Nesse exemplo, apenas a propriedade `Bold` (**Negrito**) é trabalhada. Investigue as demais e teste em seus controles, não apenas o `Label`.

- **Propriedade:** ForeColor
- **Valor para atribuir:** Blue

A propriedade `ForeColor` é responsável pela definição da cor em que o texto será exibido no controle. É uma propriedade comum aos controles visuais. Ao selecioná-la, uma lista de categorias de cores é exibida, permitindo a seleção da desejada. No exemplo, a categoria selecionada foi **Web** e a cor **Blue**.

Propriedades para o TextBox do Total

- **Propriedade:** BackColor
- **Valor para atribuir:** Yellow

A propriedade `BackColor` é responsável pela definição da cor de fundo do controle. Sua seleção é semelhante à oferecida para a propriedade `ForeColor`.

- **Propriedade:** Enabled
- **Valor para atribuir:** False

A propriedade `Enabled` é responsável por definir se o controle está ou não habilitado para interação com o usuário, seja por mouse ou teclado.

- **Propriedade:** Font (Bold e Italic)
- **Valor para atribuir:** True para Bold e Italic

A resolução para o problema proposto e a implementação do método que executa o evento `Click` do botão são exibidas a seguir.

Resolução 1.3: evento Click para o Button de cálculo do total da conta

```
private void button1_Click(object sender, EventArgs e) {
    txtTotalDaConta.Text = (Convert.ToDouble(
    txtDespesa.Text) * 1.10).ToString("N");
}
```

Toda a implementação ocorre em apenas uma linha. A classe `Convert` é novamente usada, porém, com o método `ToDouble()`, pela característica do valor a ser exibido. A chamada do `ToString()` envia agora um argumento, "`N`", que formata a `string` para um valor numérico.

A **formatação de strings** é uma prática comum, rápida e eficiente para formatar valores por meio da chamada ao método `ToString()`. Inicialmente, os separadores de centenas e decimais são os atribuídos nas **Configurações Regionais** de seu sistema operacional; porém, isso pode ser configurado.

> A MSDN oferece uma vasta documentação sobre isso, sendo um link importante: http://msdn.microsoft.com/en-us/library/dwhawy9k.aspx (relativo à **Standard Numeric Format Strings**).

1.5 Conclusão

Este capítulo apresentou três problemas para resolução e implementação por meio do Visual Studio 2013. Cada um gerou um projeto e todos ficaram alocados em uma solução. Cada projeto permitiu apresentação de controles, técnicas para resolução, conceitos e dicas sobre a plataforma, e IDE e linguagem.

Os problemas desenvolvidos possuem a característica de uma estrutura sequencial, na qual o fluxo de execução é um só. As resoluções propostas trabalharam controles, componentes e algumas de suas propriedades, eventos e alguns recursos da C#. Esses conteúdos subsidiam um avanço no conhecimento do ambiente e da linguagem, sendo possível elaborar novos projetos com estruturas semelhantes.

Problemas mais complexos serão apresentados no próximo capítulo, no qual a estrutura condicional permitirá uma maior complexidade para suas resoluções. Por mais simples que os problemas mostrados sejam, foi possível conhecer e aplicar alguns recursos do Visual Studio e conhecer um pouco da linguagem C#.

Capítulo 2

Evoluindo na complexidade: estrutura condicional

Buscando fornecer novos recursos disponíveis para os formulários Windows e para a linguagem C#, este capítulo apresenta problemas cuja resolução pode depender da resposta a uma dada pergunta ou uma avaliação de uma determinada situação ou expressão, direcionando sua execução para um caminho ou outro. Esse direcionamento é implementado por uma estrutura específica de código, conhecida como **estrutura condicional**, ou ainda, **estrutura de seleção**. As instruções responsáveis pela sua codificação serão apresentadas conforme sua utilização nos exemplos.

Uma **estrutura condicional** é uma implementação, na qual instruções, que representam expressões de avaliação das relações entre operadores, direcionam a execução do fluxo de controle da aplicação. Isso ocorre sempre de

acordo com os resultados da avaliação realizada. A estrutura condicional está sempre associada às expressões com operações relacionais entre um ou dois operadores.

2.1 Tomando uma decisão com base em uma idade identificada

Como primeiro problema a ser resolvido neste capítulo, teremos uma escola de natação que matricula seus alunos em turmas de acordo com sua categoria relacionada à faixa etária de cada um, conforme a lista a seguir.

Faremos um algoritmo para calcular a idade dos alunos que serão matriculados, identificando a qual categoria ele pertence.

a) Infantil A: 5 a 7 anos;

b) Infantil B: 8 a 10 anos;

c) Juvenil A: 11 a 13 anos;

d) Juvenil B: 14 a 17 anos;

e) Adulto: Maiores de 18 anos.

Na implementação da solução para esse problema, alguns pontos novos serão trabalhados. Inicialmente, para facilitar, serão solicitados apenas o ano de nascimento do aluno e o do seu último aniversário. A data do último aniversário não poderá ser informada antes da data de nascimento, que deverá obviamente ser inferior à de seu último aniversário. Com base nessa pré-análise, verificam-se alguns eventos que deverão ser capturados (digitação da data do último aniversário antes da de nascimento) e diversas condições, como verificação de ordem nos anos informados e a identificação da categoria, com base na idade obtida.

A figura 2.1 representa a interface desejada, que será implementada para o problema proposto nesta seção.

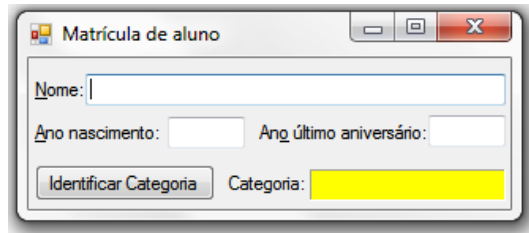



Fig. 2.1: Interface para o problema de cálculo de idade e identificação de categoria

Crie uma nova solução para os projetos que serão desenvolvidos neste capítulo. Nessa solução, crie um projeto `Windows Forms Application`. Nele, crie um formulário específico para a implementação da janela da figura 2.1.

A caixa ao lado do `Label` "Categoria", por mais que pareça um `TextBox`, é na realidade um `Label` com borda. Observe também que os `Labels` possuem uma letra sublinhada. São conhecidas como teclas de atalho, pois quando combinadas com a tecla `Alt`, levam a interação (o cursor, ou ainda o foco da aplicação) para o controle imediatamente após o `Label` na ordem de tabulação. As propriedades específicas e novas para os controles desse projeto são apresentadas na sequência.

Propriedades para o Label

- **Propriedade:** Text
- **Valor para atribuir:** &Nome:

Note o valor a ser atribuído na propriedade `Text`, o `&` antes da letra **N**. Esse símbolo causará a exibição da letra seguinte a ele, como se fosse com um sublinhado. Esta representação dá à letra marcada a funcionalidade de um "atalho" – quando combinada com a tecla `Alt` – para o controle imediatamente após o `Label` em questão, de acordo com a ordem de tabulação.

- **Propriedade:** UseMnemonic

- **Valor para atribuir:** True

A propriedade `UseMnemonic` é responsável por definir se, caso na propriedade `Text` de um `Label` exista o símbolo `&`, a letra que esse símbolo precede servirá de atalho – quando combinada com a tecla `Alt` – para o controle imediatamente após o referido, tendo a propriedade `TabIndex` como classificador desta ordem.

- **Propriedade:** BorderStyle
- **Valor para atribuir:** Fixed3D

A propriedade `BorderStyle` é responsável por definir uma borda para o `Label`. Normalmente, um `Label` não possui bordas.

- **Propriedade:** TextAlign
- **Valor para atribuir:** MiddleCenter

- **Propriedade:** Autosize
- **Valor para atribuir:** Faflse

Essa propriedade permite o redimensionamento automático, com base no tamanho da fonte.

- **Propriedade:** BackColor
- **Valor para atribuir:** Yellow

Tendo a configuração visual sido terminada, a primeira funcionalidade que será implementada é a responsável por não permitir que o ano do último aniversário seja digitado sem que o ano do nascimento o seja. Para isso, é preciso capturar o evento que ocorre quando o controle recebe o foco. Esse evento é o `Enter`. A implementação para o método que recebe a delegação de atender a ele é exibida a seguir.

Resolução 2.1: evento Enter para o TextField que recebe o ano do último aniversário do aluno

```
private void txtAnoUltimoAniversario_Enter(object sender,
       EventArgs e) {
    if (txtAnoNascimento.Text.Trim().Length < 4) {
        MessageBox.Show("É preciso informar o ANO DE " +
            "NASCIMENTO com 4 digitos", "Atenção!",
            MessageBoxButtons.OK,
            MessageBoxIcon.Information);
        txtAnoNascimento.Focus();
    }
}
```

A instrução `if()`, apresentada anteriormente, é uma instrução avaliadora; uma expressão lógica, que avalia uma determinada situação/condição. A condição aqui avaliada verifica se a quantidade de caracteres digitados no campo que representa o ano de nascimento do aluno é 4.

Para a avaliação dessa expressão foram usados: o método `Trim()` na propriedade `Text` do `TextBox`; a propriedade `Length`, que toda string possui; e o operador relacional `<` (menor que). A operação se dá apenas sobre o resultado de `Length` em comparação com a constante 4. Ou seja, `Trim()` opera sobre `Text` e retorna uma `string`, que fornece `Length`.

Tendo a avaliação da expressão retornado um valor positivo (verdadeiro), significa que o ano informado pelo usuário não possui 4 dígitos, ou não foi informado. Dessa maneira, é importante que a aplicação informe isso ao usuário e que o redirecione – ou seja, envie o foco – para o campo que precisa do dado informado de maneira correta. Assim, a informação do erro/advertência é realizada pelo método `Show()` já conhecido, e o foco é redirecionado pela chamada ao método `Focus()` do controle que o deverá receber.

A instrução `if()` é responsável pela implementação da estrutura condicional conhecida. em pseudocódigo, como **SE**. É possível inserir uma única expressão como argumentos para a instrução ou várias. No caso de mais de uma expressão, elas precisam ser separadas por um operador lógico: `&&` ou `||`, que representam, respectivamente, **E** e **OU**.

Caso apenas uma instrução seja executada, no caso de todo o `if()` retornar verdadeiro, não há necessidade dos delimitadores de bloco `{ }`. Porém, como boa prática, é bom utilizar sempre. Em conjunto com o `if()`, é possível usar as instruções `else` (se não) e `else if()` (se não se).

O método `Trim()`, pertencente à classe `String`, remove – com a assinatura apresentada na resolução – os espaços em branco do início e do final da `string` em que ele é invocado. Após esse procedimento, ele retorna uma nova `string`, sem os espaços. Existe uma segunda assinatura para o método, que permite remover um array de caracteres específicos.

A propriedade `Length`, existente em qualquer objeto `string`, retorna um valor numérico inteiro com o comprimento da `string`, ou seja, a quantidade de caracteres que ela possui.

Os operadores relacionais estabelecem, como o próprio nome diz, uma relação entre seus operandos. Esses operadores são: == (igual a), < (menor que), > (maior que), <= (menor e igual a), >= (maior e igual a), != (diferente de) e o operador unário `!` (não/negação). Toda operação relacional retorna um valor lógico, verdadeiro ou falso.

O método `Focus()` é responsável por atribuir o foco da aplicação (cursor e interação) para o controle em que é invocado. Ele retorna `true` caso consiga direcionar o foco, e `false` caso contrário.

Outra preocupação relacionada à validação dos dados é a garantia de que o ano de nascimento seja inferior ao ano do último aniversário. Para isso, é preciso identificar o método que é responsável por validar o que foi digitado. A validação tem seu começo no momento em que o controle perde o foco. Quando isso acontece, há a ocorrência do evento `Validating`. A implementação pode ser verificada na sequência.

Resolução 2.2: evento Validating para o TextField que recebe o ano do último aniversário do aluno

```
private void txtAnoUltimoAniversario_Validating(object
        sender, CancelEventArgs e) {
    if (Convert.ToInt32(txtAnoUltimoAniversario.Text) <=
            Convert.ToInt32(txtAnoNascimento.Text)) {
        MessageBox.Show("O ANO DO ÚLTIMO ANIVERSÁRIO " +
```

```
            "deve ser superior ao do ANO DE NASCIMENTO.",
            "Atenção!",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        e.Cancel = true;
    }
}
```

No código implementado para o evento `Validating`, a única instrução ainda não trabalhada é a `e.Cancel = true`. Essa instrução atribui um valor que informa que a validação deve ser interrompida, cancelada, não permitindo que o foco saia do controle que está sendo validado.

`Cancel` é uma propriedade da classe `CancelEventArgs`, representada pelo objeto `e`. É comum métodos que recebem delegação de eventos receberem argumentos. Normalmente, esses argumentos são enviados pelo evento e, em alguns casos, além de trabalhar internamente com os dados recebidos dentro do método, a alteração deles pode implicar no comportamento do método e interação do usuário com o controle, como no exemplo apresentado.

Neste ponto, vamos avaliar um pouco o código proposto; para isso, execute sua aplicação. Digite 2000 no ano de nascimento, 2008 no ano de último aniversário e, depois, 2013 no ano de nascimento. Se tudo ocorrer bem, o que era proibido foi possível de ser realizado. Ou seja, o ano de último aniversário ficou menor que o de nascimento.

Talvez, o primeiro pensamento seja implementar o evento `Validating` do controle do ano de nascimento, da mesma maneira que foi feito para o controle de ano de último aniversário. Funcionaria, mas seria redundância de código. Uma solução seria capturar o mesmo método para os dois controles. Isso é possível, pois são controles do mesmo tipo e o evento é o mesmo. Como fazer? Selecione o `TextBox` do ano de nascimento e, no evento `Validating` dele, não dê duplo clique na área em branco, mas selecione o evento já implementado. Veja na figura a seguir.

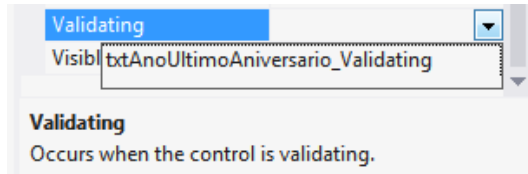



Fig. 2.2: Selecionando um método já implementado para a captura de um evento

Ao executar a aplicação, informar um valor para ano de nascimento e pressionar a tecla `Tab` causará um erro, pois o ano do último aniversário não possui ainda valor, o que gera problemas na conversão de uma `string` vazia para um `Int32`. Como resolver? Esta validação só deverá ser aplicada quando um valor já tiver sido informado para o ano do último aniversário. Dessa maneira, é preciso mudar a expressão do `if()` para que valide isso. Altere para:

```
if (txtAnoUltimoAniversario.Text != String.Empty &&
      Convert.ToInt32(txtAnoUltimoAniversario.Text) <=
      Convert.ToInt32(txtAnoNascimento.Text))
```

Observe nesta nova instrução a inserção de uma nova operação, a que verifica se a propriedade `Text` do ano do último aniversário não está vazia, sem valor informado (`!=` que significa **diferente de**). Para essa verificação, utiliza-se a propriedade `Empty` de `String`. Agora, a expressão passa a ser composta, possuindo duas operações relacionais, então é preciso que elas sejam avaliadas. Para isso, faz-se o uso do operador lógico `&&` (E), que avalia a primeira operação "e". A avaliação da segunda operação só ocorrerá caso o resultado obtido na primeira seja verdadeiro.

Os **operadores lógicos** servem para avaliar mais de uma operação. Estes operadores são o `&&` (E), `||` (OU) e `^` (OU exclusivo).

Concluído o processo de validação dos valores relacionados aos anos de nascimento e de último aniversário, é preciso identificar a qual categoria o aluno será atribuído. Este processo será implementado no evento `Click` do `Button`, e pode ser verificado na sequência.

Resolução 2.3: evento Click para o Button

```
private void btnIdentificarCategoria_Click(object sender,
                                          EventArgs e) {
    if (txtNome.Text == String.Empty ||
            txtAnoNascimento.Text == String.Empty ||
            txtAnoUltimoAniversario.Text == String.Empty) {
        MessageBox.Show("Todos os dados " +
            "solicitados devem ser informados.",
            "Atenção!", MessageBoxButtons.OK,
            MessageBoxIcon.Information);
    } else {
        int idade =
            Convert.ToInt32(txtAnoUltimoAniversario.Text) -
            Convert.ToInt32(txtAnoNascimento.Text);
        if (idade > 17) {
            lblCategoria.Text = "Adulto";
        } else if (idade > 13) {
            lblCategoria.Text = "Juvenil B";
        } else if (idade > 10) {
            lblCategoria.Text = "Juvenil A";
        } else if (idade > 7) {
            lblCategoria.Text = "Infantil B";
        } else if (idade >= 5) {
            lblCategoria.Text = "Infantil A";
        } else {
            lblCategoria.Text = "Não existe categoria";
        }
    }
}
```

A estrutura condicional implementada pela instrução `if()` aparece de uma maneira mais completa no exemplo. Inicialmente, verifica-se se todos os dados obrigatórios foram informados e, caso algum deles não cumpra esse requisito, nada é processado. Passando positivamente por essa validação, é obtida a idade do aluno, e essa idade é avaliada por uma série de expressões disjuntas. Na identificação da categoria, um texto é atribuído ao `Label` que a exibirá.

Normalmente, quando se realiza o cadastro de uma pessoa, o dado relacionado à idade parte de uma data de nascimento. O comum é solicitar a data de nascimento completa, e não apenas o ano. Entretanto, obter a diferença entre duas datas não envolve apenas uma operação de subtração.

Para realização desse cálculo e de outros que envolvam data, são usados recursos oferecidos pela linguagem (classes e métodos). A figura 2.3 traz uma nova versão para a resolução do problema da **Resolução 2.1**.

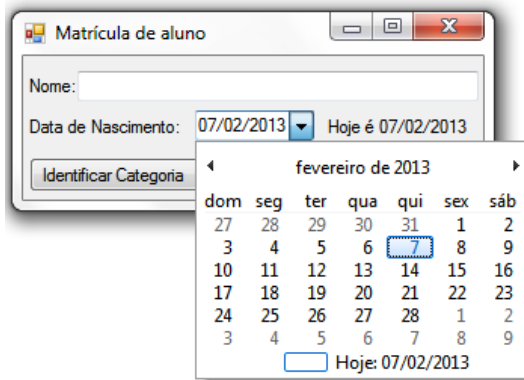



Fig. 2.3: Utilização de datas e controle para calendário

É possível verificar na nova janela duas mudanças: um calendário (`DateTimePicker`) e um `Label` que apresenta a data atual. Para que a data atual seja exibida na janela no momento em que ela é exibida, foi implementada essa operação no construtor da classe do formulário.

Resolução 2.4: construtor da classe do formulário

```
public FormCategoriaPorIdadeV2() {
    InitializeComponent();
    lblHoje.Text = "Hoje é " +
        DateTime.Now.ToShortDateString();
}
```

A classe `DateTime` representa as datas e horas que estejam no intervalo de 00h:00m de 1 de janeiro de 1601 até 31 de dezembro de 9999, em um calen-

dário conhecido como *Anno Domini* (AD). Estes limites podem ser obtidos pelas propriedades (de leitura) `MinValue` e `MaxValue`. Diversos métodos são oferecidos para manipular datas e horas, assim como propriedades para consultar e configurar.

A propriedade `Now` retorna a data e hora atual com base na localização configurada em seu sistema operacional. Existem outras diversas propriedades que podem ser utilizadas, como: `Date`, `Day`, `DayOfWeek`, `DayOfYear`, `Hour`, `Kind`, `MiliSecond`, `Minute`, `Month`, `Second`, `Ticks`, `TimeOfDay`, `Today`, `UtcNow` e `Year`.

O método `ToShortDateString()` formata a data para a máscara `dd/MM/aaaa`, ou a que tenha sido configurada no sistema operacional. Essa máscara exibe o dia, mês e ano, com valores numéricos.

Assim como existem diversas propriedades para `DateTime`, existem diversos métodos que podem ser invocados para o retorno da propriedade `Now`. Cabe uma identificação deles e uma investigação mais detalhada.

O `DateTimePicker` é um controle que permite ao usuário a seleção de uma data específica, diretamente em um calendário. Ele possibilita a navegação entre meses, anos e semanas, dentre diversas outras funcionalidades.

Uma vez que a janela tenha sido alterada para a seleção da data de nascimento e exibição da data atual, é preciso alterar a implementação do método que captura o evento `Click`, que está a seguir.

Resolução 2.5: trecho com novo código para obtenção da idade

```
private void btnIdentificarCategoria_Click(object sender,
                                           EventArgs e) {
// Trecho para validação do nome continua o mesmo e
// foi omitido aqui

    TimeSpan tsQuantidadeDias = DateTime.Now.Date -
        dtpDataDeNascimento.Value;
    int idade = (tsQuantidadeDias.Days / 365);

// Considerar o ano com 365 dias pode não resultar
// em um valor preciso
```

```
// Trecho para identificação das categorias continua o
// mesmo e foi omitido aqui
}
```

Por meio da subtração das datas, armazenando o resultado em um objeto da classe `TimeSpan`, é possível obter e trabalhar com os resultados relacionados à diferença desejada. Essa classe fornece diversas propriedades, entre elas a `Days`, que possui a quantidade de dias que existem entre as duas datas operadas. Após essa operação, uma simples divisão retorna, de maneira aproximada, a quantidade de anos a que se referem os dias obtidos. Esse valor é aproximado pelo fato de não serem levados em consideração os anos bissextos.

Quando o resultado de uma divisão é atribuído a uma variável do tipo `int`, apenas a parte inteira (o quociente) é armazenada, sendo descartado o resto (parte fracionária).

Agora que os conhecimentos para obter o quociente e resto de uma divisão foram apresentados, procure obter a idade completa de uma pessoa, em anos, meses e dias. Lembre-se apenas de que considerar sempre um ano com 365 dias e um mês com 30 dias não resultará em um resultado preciso.

A implementação apresentada trouxe também, pela primeira vez, o uso de comentários. As barras (`//`) identificam a linha como comentário, dispensando a compilação nela. É possível também inserir **comentário** em uma instrução finalizada com ponto e vírgula. Existem outros meios de fazer uso de comentários, iniciando com `/*` e terminando com `*/`, assim o comentário será em várias linhas. Caso queira que o comentário possa gerar documentação em XML, basta usar `///` em cada linha comentada.

2.2 Identificando o peso ideal para uma pessoa

Dando sequência com um novo problema, desenvolva uma aplicação que, ao solicitar ao usuário a altura e sexo de uma pessoa, seja possível informar como resultado o peso ideal para os dados informados. Para identificação desse peso, utilize as fórmulas a seguir:

- **Para homens:** (72.7*h)-58

- **Para mulheres:** (62.1*h)-44.7

Pela leitura do enunciado, constata-se que a solução para o problema é constituída por: uma situação de leitura de valores informados pelo usuário; realização de uma análise sobre estes dados; e a tomada de decisões, tendo como base o resultado dessa análise. Este é o cenário típico para problemas baseados em condições, no qual se aplica uma estrutura de seleção para a sua resolução.

Com base nessa situação, a aplicação desenvolvida para esse problema faz uso de novos controles, muito comuns para problemas com essa característica. Também serão apresentadas novas técnicas e recursos.

A figura 2.4 representa a interface desejada, que será implementada para o problema proposto nesta seção.

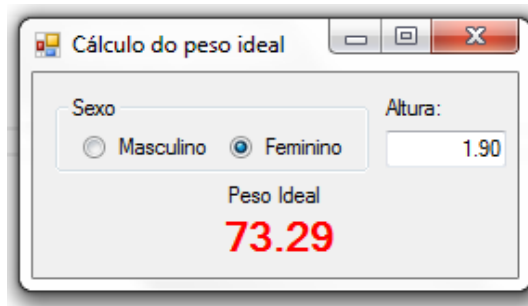



Fig. 2.4: Interface para o problema de cálculo de peso ideal

Verifica-se na figura que representa a aplicação, que não existe nenhum botão para que o usuário dispare o processamento desejado. Dois novos controles também podem ser identificados: um *container*, com o título `Sexo` e, dentro deste, dois `RadioButtons`. O resultado desejado, que é o peso ideal, pode ser visualizado na base da janela, com uma fonte maior, em negrito e cor diferenciada.

Um `container`, em aplicações visuais, é um controle que armazena dentro de si outros controles. Até o momento, apenas o formulário foi visto como um container. Entretanto, é comum em uma janela o agrupamento de

controles, de acordo com as suas características, como é o caso agora do uso do `GroupBox`.

O `GroupBox` é um container de outros controles. Normalmente, dentro deste controle, são inseridos controles do tipo `RadioButton`. A união deles permite que diversas opções (`RadioButtons`) possam ser inseridas no `GroupBox`, com a garantia de que apenas uma delas estará selecionada (checada).

Na `Toolbox`, na categoria `Containers`, selecione e arraste o controle `GroupBox` e posicione-o corretamente no formulário. Na categoria de controles `Common Controls` da `Toolbox`, selecione e arraste o controle `RadioButton` duas vezes, dentro do `GroupBox`. As propriedades a serem configuradas para esses controles são a `Name` e `Text`. Configure-as com base na imagem da figura 2.4.

Com o formulário criado e configurado, é preciso definir e implementar o comportamento desejado. Para obtenção do peso ideal, como visto na fórmula, além da altura, é preciso saber o sexo da pessoa. Essa resposta, visualmente, é facilmente obtida, bastando verificar qual `RadioButton` estará **checado**. Entretanto, "programaticamente", para se obter esta resposta, é preciso verificar o estado de cada um deles; se está checado, ou não. Essa verificação é realizada na propriedade `Checked`, onde `True`, o `RadioButton` está checado, e `False` não.

Como um `Button` não foi inserido no desenho da janela, não existe, a princípio, uma ação que disparará todo o processamento, começando por esta verificação. Entretanto, a técnica adotada nesse exemplo é que o processamento se dará por eventos disparados pela alteração do **Sexo** e da **Altura**. Sendo assim, a classe do formulário manterá um atributo que será o responsável em manter sempre atualizado o sexo informado. A seguir, apresento essa declaração.

Resolução 2.6: trecho de declaração da variável do RadioButton selecionado

```
namespace ProjetoPesoIdeal {
    public partial class CalculoDePesoIdeal : Form {
        RadioButton rbnSelecionado = null;
```

```
        // Código restante omitido
    }
}
```

A variável `rbnSelecionado` será responsável por ter, de maneira atualizada, o `RadioButton` que esteja checado (propriedade `Checked` com o valor `True`). Essa atribuição será realizada pelo método que recebe a delegação do evento `CheckedChanged`.

Resolução 2.7: evento CheckedChanged para os RadioButtons

```
private void rbnMasculino_CheckedChanged(object sender,
        EventArgs e) {
    RadioButton rbn = (RadioButton) sender;
    if (rbn.Checked) {
        rbnSelecionado = rbn;
        SetPesoIdeal();
    }
}
```

O evento `CheckedChanged` é disparado quando há alguma mudança na propriedade `Checked`. Dessa maneira, é possível saber quando um `RadioButton` muda para os estados checado ou não checado.

Todos os eventos apresentados até agora trazem em sua assinatura um argumento especial, o `object sender`. Esse objeto traz para o método o controle (ou objeto) que causou o evento. Este recurso é muito útil, pois permite que um mesmo método atenda a um mesmo evento de diversos controles, como demonstrado na figura 2.2.

Com esta técnica, independente de qual controle tenha disparado o evento, o método sempre saberá qual foi o que causou o evento. Dessa maneira, a primeira linha do método, onde o `sender` – após uma operação de *cast* – é atribuído à variável `rbn`, garante que essa variável terá sempre o controle que disparou o evento.

O *cast* é uma operação que "força", sempre que possível, que um objeto de determinado tipo seja convertido para outro tipo, para sua manipulação. No

exemplo, um objeto (`sender`) declarado como `object`, para ser atribuído a uma variável do tipo `RadioButton`, precisou ser explicitamente convertido para o tipo de destino.

A instrução `if()` avalia, por meio da propriedade `Checked`, se o `RadioButton` que causou o evento está checado. Caso ele esteja, o campo da classe `rbnSelecionado` recebe a instância do que disparou o evento. Após essa atribuição, o método `SetPesoIdeal()` é invocado. Esse método pode ser visualizado na sequência.

Resolução 2.8: método SetPesoIdeal()

```
private void SetPesoIdeal() {
    try {
        double altura = Convert.ToDouble(txtAltura.Text);
        double pesoIdeal;
        if (rbnSelecionado.Text.Equals("Masculino"))
            pesoIdeal = (72.7 * altura) - 58;
        else
            pesoIdeal = (62.1 * altura) - 44.7;
        lblPesoIdeal.Text = pesoIdeal.ToString("N");
    }
    catch (Exception e) {
        MessageBox.Show("Selecione o sexo e informe a
           altura corretamente", "Atenção!",
           MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

Existem outras técnicas para identificação de qual `RadioButton` está checado dentro de um `GroupBox`. Uma delas faz uso do *Language INtegrated Query* (LINQ) e expressões Lambda, excelentes recursos oferecidos pelo .NET. O LINQ e expressões Lambda não serão tratados em detalhes neste ponto, mas serão apresentados mais à frente no capítulo 7. Com o LINQ e Lambda, em vez de ter um atributo da classe que é atualizado a cada alteração de item selecionado, apenas a linha seguinte seria suficiente:

```
rbnSelecionado = gbxSexo.Controls.OfType<RadioButton>().
SingleOrDefault(r => r.Checked == true);
```

A comparação de `r.Checked == true` é ilustrativa, pois como `Checked` é booleano, bastaria utilizar `r.Checked`.

O método `Equals()` é um método virtual, que pode ser sobrescrito por qualquer classe. Ele busca, por meio de atributos – ou campos, como são chamados em C# –, verificar se o objeto que invoca o método é igual ao objeto enviado como argumento para ele. O operador de igualdade (==) é indicado quando se busca comparar valores que têm seu armazenamento por valor, e não referência. Quando o operador == for utilizado entre dois objetos, o que ocorre é a comparação entre as referências deles, ou seja, se ocupam o mesmo endereço de memória. Dessa maneira, sempre que houver necessidade de comparação entre objetos armazenados por referência, ela deve ocorrer por meio do método `Equals()`.

Existem alguns novos recursos e técnicas nesta implementação. A primeira é a justificativa de um método específico. Por que não resolver no método delegado pelo evento? Bem, o cálculo do peso não será realizado por um evento de um único tipo de controle. Além do `CheckedValue` dos `RadioButtons`, o evento `TextChanged` do `TextBox`, referente à altura, também causará a necessidade do cálculo. A seguir, apresento o método que captura o evento `TextChanged`.

Resolução 2.9: evento TextChanged para os TextBoxs

```
private void txtAltura_TextChanged(object sender,
        EventArgs e) {
    SetPesoIdeal();
}
```

O evento `TextChanged` é disparado quando há alguma mudança no texto existente na propriedade `Text`. Dessa maneira, esse evento ocorre, por exemplo, a cada tecla que gera um conteúdo.

Embora as assinaturas dos dois métodos que recebem a delegação dos eventos sejam as mesmas, existem situações em que isso não ocorre. Dessa maneira, não seria possível o uso do mesmo método para eventos com assinaturas diferentes. Quando for este o caso, seria correto implementar a mesma resolução em mais de um método, ou criar um método específico e invocá-lo? As boas práticas e o paradigma da Orientação a Objeto (OO) direcionam

para a reutilização. Neste caso, a **reutilização** é empregada na criação de um método.

A **reutilização** de código é um processo durante o desenvolvimento de uma aplicação que o usuário final não vê. Ela está diretamente ligada à **manutenibilidade**, além de outras características. Esta última pode ser "notada" pelo usuário final, pois um sistema bem-analisado, modelado e projetado, aplica a reusabilidade, propiciando uma excelente manutenibilidade. Ou seja, mudanças que possam ocorrer no código, sejam estas corretivas ou evolutivas, tendem a causar um menor impacto ao usuário final, assim como são implementadas e validadas em um menor tempo.

Para efeito demonstrativo, nessa implementação não foram usados os blocos delimitadores para as instruções que serão executadas na avaliação positiva do `if()`, ou no caso do `else`. Como apenas uma instrução será executada, as chaves (`{ }`) são desnecessárias.

Outra boa prática de OO e que foi aplicada nessa resolução é o uso de **tratamento de exceções**. Essa técnica trabalha a seguinte situação: se existe a possibilidade de ocorrência de erros, e caso estas não sejam verificadas (com `if()`), o bloco é executado; e caso seja prevista a possibilidade de erros ou exceções, use a estrutura `try...catch` para prevenir, capturar e tratar esses eventos.

O **tratamento de exceções** é utilizado para capturar e tratar situações de erros que possam acontecer durante a execução de um programa. Quando uma exceção acontece, significa que algum problema foi detectado. Sendo assim, busca-se capturar a exceção ocorrida e tratá-la, para que o programa não seja interrompido de maneira brusca ou sem o tratamento devido.

Na estrutura `try...catch`, as instruções que fazem parte da cláusula `try` têm a semântica de "tentarem" executar. Caso alguma exceção ocorra, o fluxo é imediatamente desviado para a instrução `catch`, ignorando qualquer instrução seguinte da cláusula `try`.

Quando uma exceção ocorre, um objeto da classe que a representa é enviado para a cláusula `catch`, que captura essa ocorrência. Dessa maneira, é possível tratar essa exceção e permitir uma continuidade na execução, sem que ela seja interrompida, ou que uma mensagem de erro, que o usuário provavelmente não entenderá, seja exibida. Existem diversas exceções,

`Exception` é a mais generalizada.

Para concluir a implementação, atribua o método criado para evento `CheckedChanged` do `RadioButton` do sexo masculino, para o `RadioButton` do sexo feminino.

2.3 Identificando um reajuste salarial

Para finalizar os exemplos deste capítulo, desenvolva um programa que solicite ao usuário que informe o salário mínimo atual e os seguintes dados de um estagiário: o turno de trabalho, a função e o número de horas trabalhadas no mês. Com base no turno do trabalho do funcionário, deverá ser calculado um coeficiente, como segue:

- **Matutino:** 1% do salário mínimo;
- **Vespertino:** 2% do salário mínimo;
- **Noturno:** 3% do salário mínimo.

Tendo o valor do coeficiente calculado, com base nele obtenha o salário bruto do estagiário, que é o produto dele (coeficiente) e as horas trabalhadas. Sobre o salário bruto é preciso obter um imposto hipotético a ser descontado, como segue:

- Calouro: salário < 300,00 tem imposto de 1%;
- Salário >= 300,00 tem imposto de 2%;
- Veterano: salário < 400,00 tem imposto de 3%;
- Salário >= 400,00 tem imposto de 4%.

Para alguns estagiários, será também pago um valor em forma de gratificação. Para os que preencherem todos os requisitos, ela será de R$ 50,00; caso contrário, de R$ 30,00.

Os requisitos são:

- **Turno:** noturno;

- **Número de horas trabalhadas:** superior a 80.

Também é oferecido o auxílio-alimentação. Se o estagiário preencher algum dos requisitos, seu auxílio-alimentação será de um terço do seu salário bruto; caso contrário, será de metade deste terço. Os requisitos são:

- **Categoria:** calouro;
- **Salário bruto:** menor que a metade do salário mínimo.

Uma classificação deve ser atribuída ao salário líquido: bruto menos imposto, mais gratificação, mais auxílio-alimentação. Essa classificação deve seguir os requisitos a seguir:

- **Menor que R$ 350,00:** mal remunerado;
- **Entre R$ 350,00 e R$ 600,00:** normal;
- **Maior que R$ 600,00:** bem remunerado.

Para a resolução desse problema que possui vários pontos de avaliações, será feito uso de outra estrutura condicional (ou de seleção), a `switch...case`. Um controle de "auxílio" ao usuário, conhecido como `ToolTip`, e um controle que representa uma lista de valores exibidos em forma de linhas, conhecido como `ListBox`, serão introduzidos também nessa resolução.

A figura 2.5 representa a interface desejada, que será implementada para o problema proposto nesta seção.

Fig. 2.5: Interface para o problema de cálculo de um salário

Na figura que representa a aplicação, é possível verificar um componente que exibe um resultado dos valores obtidos após o processamento dos dados informados. Esse controle é o `ListBox`. Também é visível um "balão" com texto que auxilia o usuário em relação a o que informar em um determinado campo do formulário, que é o `ToolTip`. A caixa em amarelo, que representa a situação do salário do estagiário, é um `TextBox` com formatação em seu estilo visual. Os demais controles são todos já conhecidos.

Um controle `ListBox` exibe uma lista de itens. Esses itens normalmente são strings que, uma vez selecionadas, podem disparar alguma ação na aplicação. Além disso, eles são armazenados na propriedade `Items` e podem ser exibidos em uma única coluna ou em múltiplas.

Para a exibição em múltiplas colunas, à propriedade `MultiColumn` deve-se atribuir o valor `true`. A seleção dos itens nesse controle pode ser configurada para permitir que mais de um item seja selecionado. Para isso, é preciso configurar a propriedade `SelectionMode`. Caso o tamanho visual do controle (com os itens exibidos) seja inferior à quantidade dos itens tanto em altura quanto comprimento, barras de rolagem são exibidas, quando configuradas para exibição automática.

Lembre-se de consultar o site da MSDN para obter informações detalhadas e exemplos sobre os controles apresentados.

O `ToolTip` é um componente que oferece um ótimo recurso visual para exibição de mensagens auxiliadoras para os usuários. Para que o texto possa ser exibido, o usuário precisa apontar o cursor do mouse para o controle que possua uma configuração para esse recurso.

Cabe ressaltar que esse componente não é representado graficamente no formulário em tempo de implementação. Ele fica alocado em uma área específica da área de desenho de formulários, conforme retratado na figura 2.6, e as configurações são realizadas nele. Cada controle que fará uso desse recurso recebe uma nova propriedade quando o `ToolTip` é arrastado. Esta propriedade tem o nome `ToolTip on tooltip1`, sendo `tooltip1` o nome dado ao componente.

Todos os controles necessários estão na categoria `Common Controls` da `ToolBox`; arraste-os. Ao arrastar o `ToolTip`, solte-o sobre o formulário ou sobre algum controle dele. Verifique, por meio da figura 2.6, que ele fica em uma área não visual na área de desenho de formulários, como já comentado. Configure os demais controles, tendo a imagem da figura 2.5 como base.

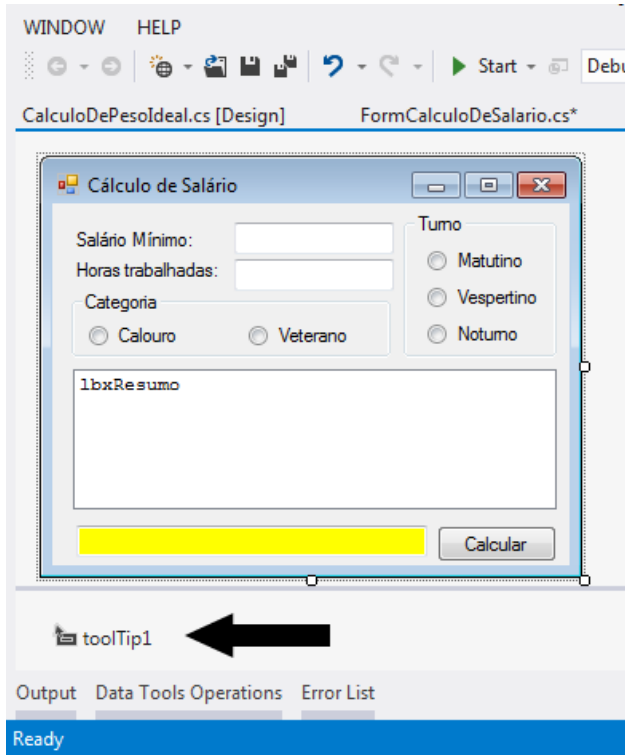



Fig. 2.6: Área de desenho de formulários com controles não visuais

Com esse controle no formulário, uma nova propriedade é inserida para os controles visuais (figura 2.7). Como ele permite configurações visuais que podem ser diferentes para grupos distintos de controles, é possível inserir quantos `ToolTips` forem necessários e, para cada um, uma nova propriedade é inserida.

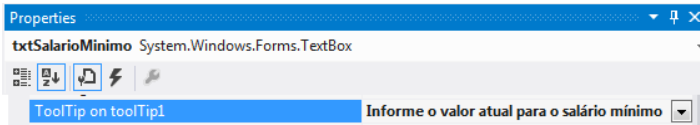



Fig. 2.7: Propriedade inserida nos controles para o ToolTip

Com exceção do `Button`, os controles terão seus valores manipulados e verificados por meio de código. Dessa maneira, nomeie-os de acordo com a sua semântica. De igual forma, configure a propriedade `Text` de cada um e configure o estilo do `TextBox` ao lado do `Button` da maneira que preferir. Para o `ListBox`, configure a fonte para `Courier`. As propriedades que precisarão de uma configuração específica são para o `ToolTip` e são apresentadas na sequência.

Sempre que precisar que todas as letras de um texto possuam o mesmo tamanho, procure escolher uma letra monoespaçada, como a `Courier`.

Propriedades para o ToolTip

- **Propriedade:** IsBallon
- **Valor para atribuir:** True

Quando essa propriedade recebe o valor `True`, o texto auxiliar aparece em um balão de diálogo em vez de em uma caixa retangular.

- **Propriedade:** ToolTipIcon
- **Valor para atribuir:** Info

Nas caixas de auxílio (ou balões) exibidos, é possível adicionar um ícone, o que dá uma boa visão semântica do tipo de auxílio que está sendo exibido. O controle oferece três possíveis opções: `Info`, `Warning` e `Error`.

- **Propriedade:** ToolTipTitle
- **Valor para atribuir:** Ajuda

Essa propriedade coloca em destaque um título para a mensagem de auxílio ao usuário.

O problema proposto neste exemplo emprega diversas verificações, com diversos processamentos, inclusive com dependências entre eles. Aliada a essa nova situação, toda a resolução proposta aqui será delegada a um método específico, que, por sua vez, terá a responsabilidade de também delegar para métodos ainda mais específicos. Ou seja, não é o método que recebe a delegação do evento que tem a responsabilidade de resolver o problema. Ele precisa saber quem resolve e, então, encaminhar para o responsável o que a interface com o usuário oferece, como parâmetros. A seguir, apresento o método para o evento `Click` do `Button`.

Resolução 2.10: evento Click para o Button

```
private void btnCalcular_Click(object sender, EventArgs e) {
    RadioButton rbnTurno = gbxTurno.Controls.OfType
        <RadioButton>().SingleOrDefault(r => r.Checked);
    RadioButton rbnCategoria = gbxCategoria.Controls.
        OfType<RadioButton>().SingleOrDefault
        (r => r.Checked);

    RealizarProcessamento(rbnTurno, rbnCategoria,
        Convert.ToDouble(txtHorasTrabalhadas.Text),
        Convert.ToDouble(txtSalarioMinimo.Text));
}
```

O processamento para obter os valores desejados é delegado a um método chamado `RealizarProcessamento()`. Para cumprir seu objetivo, ele precisa saber quais opções estão ativas – ou qual `RadioButton` possui a propriedade `Checked` igual a `true` – nos containers do tipo `GroupBox`. Dessa maneira, esses valores são obtidos e enviados como argumento para esse método, que tem seu código apresentado na sequência.

Resolução 2.11: método RealizarProcessamento()

```
private void RealizarProcessamento(RadioButton rbnTurno,
        RadioButton rbnCategoria,
```

```
        double horasTrabalhadas,
        double valorSalarioMinimo) {
    double valorCoeficiente = GetCoeficiente(rbnTurno);
    double valorGratificacao =
        GetGratificacao(rbnTurno, horasTrabalhadas);
    double valorSalarioBruto = horasTrabalhadas *
        valorCoeficiente;
    double valorImposto = GetValorImposto(rbnCategoria,
        valorSalarioBruto);
    double valorAuxilioAlimentacao =
        GetValorAuxilioAlimentacao(rbnCategoria,
            valorSalarioBruto, valorSalarioMinimo);
    double valorSalarioLiquido = (valorSalarioBruto +
        (valorGratificacao + valorAuxilioAlimentacao)) -
        valorImposto;
    ApresentarResultados(valorCoeficiente,
        valorSalarioBruto, valorImposto,
        valorGratificacao, valorAuxilioAlimentacao,
        valorSalarioLiquido);
}
```

Observe na implementação do método `RealizarProcessamento()` que alguns valores são obtidos por meio de invocação a outros métodos. Esta técnica é importante, pois não traz para um método responsabilidades diferentes das que ele efetivamente tem. Após a realização de todos os cálculos, os valores obtidos são apresentados também por um método específico. A seguir, trago a implementação para o método `GetCoeficiente()`.

Resolução 2.12: método GetCoeficiente()

```
private double GetCoeficiente(RadioButton rbnTurno) {
    double valorCoeficiente = 0;
    switch (rbnTurno.Text) {
        case "Matutino":
            valorCoeficiente = Convert.ToDouble(
                txtSalarioMinimo.Text) * 0.01;
            break;
        case "Vespertino":
```

```
            valorCoeficiente = Convert.ToDouble(
                txtSalarioMinimo.Text) * 0.02;
            break;
        case "Noturno":
            valorCoeficiente = Convert.ToDouble(
                txtSalarioMinimo.Text) * 0.03;
            break;
    }
    return valorCoeficiente;
}
```

Todo o processamento realizado nesse método é de fácil leitura. Entretanto, a estrutura condicional adotada, `switch()`, aparece pela primeira vez. Essa estrutura trabalha sempre em relação a um valor avaliado, no caso o `rbnTurno.Text`. O valor contido nessa variável será comparado com as constantes de cada `case`. A verificação que for verdadeira terá todo o código abaixo dela executado.

Devido a essa situação, cada conjunto de instruções atribuído a um `case` possui uma instrução `break`. Para exemplificar, no caso do valor de `rbnTurno.Text` ser "`Matutino`", e a não existência do `break`, todas as instruções abaixo seriam executadas, até que um `break` fosse encontrado ou o `switch()` finalizado.

A documentação de referência define a estrutura condicional `switch()` como uma instrução de controle que lida com várias seleções, passando o controle para uma das demonstrações de casos dentro de seu corpo. A execução é transferida para o `case`, cuja expressão constante combina com o valor recebido como argumento para o `switch()`. Ela pode incluir qualquer número de `case`, mas dois `case`s não podem ter o mesmo valor constante. A execução do bloco de instruções começa na primeira instrução do `case` selecionado e continua até que a instrução `break` seja encontrada. Essa instrução de salto (`break`) é necessária para cada `case`, inclusive o último (MSDN).

Na sequência, há o método `GetGratificacao()`. Nele, são realizadas verificações e alguns processamentos, algo já trabalhado no livro, portanto não exige maiores explicações.

Resolução 2.13: método GetGratificacao()

```
private double GetGratificacao(RadioButton rbnTurno,
        double horasTrabalhadas) {
    double valorGratificacao = 30;
    if (rbnTurno.Text.Equals("Noturno") &&
            horasTrabalhadas > 80)
        valorGratificacao = 50;
    return valorGratificacao;
}
```

O método `GetValorImposto()` é representado a seguir. Nessa implementação, há o uso das duas estruturas condicionais em conjunto.

Resolução 2.14: método GetImposto()

```
private static double GetImposto(RadioButton rbnCategoria,
        double valorSalarioBruto) {
    double valorImposto = 0;
    switch (rbnCategoria.Text) {
        case "Calouro":
            if (valorSalarioBruto < 300)
                valorImposto = valorSalarioBruto * 0.01;
            else
                valorImposto = valorSalarioBruto * 0.02;
            break;
        case "Veterano":
            if (valorSalarioBruto < 400)
                valorImposto = valorSalarioBruto * 0.03;
            else
                valorImposto = valorSalarioBruto * 0.04;
            break;
    }
    return valorImposto;
}
```

Para obtenção do valor do auxílio-alimentação, é invocado o método `GetAuxilioAlimentacao()`, apresentado na sequência.

Resolução 2.15: método GetAuxilioAlimentacao()

```
private double GetAuxilioAlimentacao(RadioButton
        rbnCategoria, double valorSalarioBruto,
        double valorSalarioMinimo) {
    double valorAuxilioAlimentacao =
        (valorSalarioBruto / 3) / 2;
    if (rbnCategoria.Text.Equals("Calouro") &&
            (valorSalarioBruto <
            (valorSalarioMinimo / 2)))
        valorAuxilioAlimentacao = (valorSalarioBruto / 3);
    return valorAuxilioAlimentacao;
}
```

Os dados processados precisam ser apresentados ao usuário final. Para isso, devem ser responsabilidade de um método específico. Neste sentido, o método `ApresentarResultados()` formata e apresenta os resultados no controle `ListBox`, conforme apresentado na sequência.

Resolução 2.16: método ApresentarResultados()

```
private void ApresentarResultados(double valorCoeficiente,
        double valorSalarioBruto, double valorImposto,
        double valorGratificacao, double
        valorAuxilioAlimentacao, double
        valorSalarioLiquido) {
    txtSituacaoEstagiario.Text = GetSituacaoEstagiario(
        valorSalarioLiquido);
    lbxResumo.Items.Add(String.Format("{0,-29}{1,12:C}",
        "Valor do coeficiente:", valorCoeficiente));
    lbxResumo.Items.Add(String.Format("{0,-29}{1,12:C}",
        "Salário bruto:", valorSalarioBruto));
    lbxResumo.Items.Add(String.Format("{0,-29}{1,12:C}",
        "Valor do imposto :", valorImposto));
    lbxResumo.Items.Add(String.Format("{0,-29}{1,12:C}",
        "Valor da gratificação :", valorGratificacao));
    lbxResumo.Items.Add(String.Format("{0,-29}{1,12:C}",
        "Valor auxilio alimentação :",
```

```
        valorAuxilioAlimentacao));
    lbxResumo.Items.Add(String.Format("{0,-29}{1,12:C}",
        "Salário líquido:", valorSalarioLiquido));
}
```

O último método a ser apresentado é o responsável pela identificação da situação do qualificador do salário do estagiário, que é o `GetSituacaoEstagiario()` e pode ser visualizado a seguir.

Resolução 2.17: método ApresentarResultados()

```
private string GetSituacaoEstagiario(
        double valorSalarioLiquido) {
    if (valorSalarioLiquido < 350)
        return "Mal remunerado";
    else if (valorSalarioLiquido < 600)
        return "Normal";
    else
        return "Bem remunerado";
}
```

Para concluir a implementação, aplique regras para que os `TextBox`s não estejam vazios para a realização do processamento.

2.4 Conclusão

Os problemas apresentados neste capítulo aplicaram uma estrutura conhecida como estrutura condicional, ou de seleção. Ela parte de uma avaliação, que pode ser baseada em uma ou várias condições.

Três enunciados foram propostos e, em cada um deles, novas técnicas, recursos da linguagem e controles foram apresentados e trabalhados. Ressalta-se que a linguagem C# oferece duas categorias para estrutura de seleção: o `if...else` e o `switch...case`. Os exemplos buscaram apresentar o uso isolado de cada uma delas, mas também demonstraram que elas podem trabalhar em conjunto.

Com o conhecimento adquirido, com a nova estrutura, novos controles, componentes, técnicas e recursos da linguagem, aumenta-se o horizonte de

possíveis problemas que podem ser resolvidos. O próximo capítulo trabalha a última das três estruturas básicas, que propiciará um conhecimento ainda maior da linguagem C#.

Capítulo 3

Resolvendo problemas mais complexos: estruturas de repetição

As duas estruturas de execução de um fluxo de instruções, apresentadas nos capítulos 1 e 2, trouxeram algumas técnicas e recursos que podem ser (e são) usados comumente. Entretanto, os problemas propostos trabalharam a execução desses fluxos uma única vez.

Existem situações onde um problema exige ou apresenta como resolução um conjunto de instruções que devem ser executadas por mais de uma vez. Para essas situações, é feito o uso de uma estrutura de repetição, que, assim como a condicional, possui variações de sua implementação. Este capítulo traz problemas que poderão ser resolvidos com o uso desse tipo de estrutura,

utilizando suas variações.

Uma **estrutura de repetição** é uma implementação na qual uma instrução, ou um conjunto delas, pode ocorrer mais de uma vez. Esta iteração pode se suceder um número determinado de vezes (repetição contada) ou com base em uma determinada condição (verificada no início ou ao final da estrutura). Além dessas estruturas clássicas, o C# oferece a repetição para "varrer/percorrer" uma coleção de objetos, conhecida como `for each`.

3.1 Registro do consumo de energia em um condomínio

Para iniciar os exemplos deste capítulo, assuma que, em um condomínio fechado de 10 residências, a companhia de fornecimento de energia está fazendo um levantamento de consumo. O responsável por esse levantamento informará ao seu programa a quantidade de Kw consumido (por residência), e esse programa deverá informar a quantidade de Kw que representa 20% do total consumido de cada residência. Ao final, deverá informar a projeção de consumo total do condomínio, já descontado os 20% de cada residência.

Para a resolução deste problema, será trabalhado o uso de classes de negócio. O objetivo desta técnica é trazer novos conceitos sobre Orientação a Objetos. Também será utilizaremos coleções (*collections*), que possibilitam que diversos objetos sejam armazenados em um único objeto, como um conjunto.

Em relação a controles, será apresentado o `DataGridView`, que exibe diversos itens, como uma planilha. A aplicação deverá permitir a digitação do número da casa e a quantidade consumida em Kw para a casa informada. A cada leitura realizada, o `DataGridView` deverá exibi-la, como também todas as leituras já realizadas. Uma casa não poderá ter mais de uma leitura realizada e, ao final do registro de todas as leituras, o usuário poderá realizar o processamento que informará o total consumido pelo condomínio, o total descontado e o valor com o desconto.

A figura 3.1 representa a interface desejada, que será implementada para o problema proposto nesta seção.

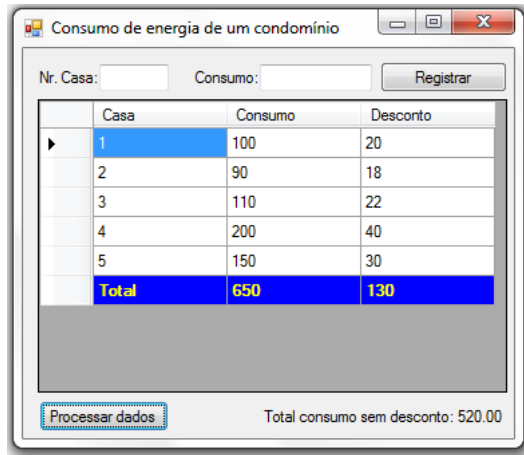



Fig. 3.1: Interface para o problema de consumo de energia

Crie uma nova solução para os projetos que serão desenvolvidos neste capítulo. Crie um projeto `Windows Application`, com um formulário específico para a implementação da janela da figura 3.1. O controle que exibe as leituras realizadas é um `DataGridView`, que está na categoria `Data`. Os demais já conhecemos: `Label`, `TextBox` e `Button`.

O `DataGridView` é um controle que permite a exibição de dados provenientes de diversas fontes, como registros de uma tabela em uma base de dados e coleções. Cada item de uma coleção (ou de um registro de tabela) é exibido como uma linha, e cada campo da tabela (ou propriedade de um objeto) é exibido em uma coluna. Existem diversos recursos que permitem personalizar o visual e operação desse controle.

Buscando introduzir sempre conceitos relacionados à Orientação a Objetos, essa nova implementação trará o uso de classes que representem a camada de modelo da aplicação. A janela possibilita ao usuário inserir informações referentes a uma leitura de consumo de energia. Dessa maneira, a leitura passa a ser uma instância ligada diretamente ao negócio da aplicação. Ela faz parte do problema. Assim, uma classe específica para prover objetos relacionados à leitura de cada casa será utilizada. A **Resolução 3.1** traz a implementação dessa classe de negócio, que nesse problema é chamada de `Leitura`.

Quando se desenvolve uma aplicação, principalmente uma que busque seguir o paradigma de Orientação a Objetos, objetiva-se ao máximo o desenvolvimento em camadas, onde cada uma delas tem sua responsabilidade e oferece meios de interação entre as demais. Um padrão conhecido e solidificado entre os desenvolvedores é o **MVC** (Modelo-Visão-Controle, do inglês *Model-View-Controller*).

Pensando neste sentido, as camadas de visão implementadas nos projetos até este momento são de fácil identificação e são compostas pelos formulários/janelas. A camada de controle pode ser vista como a classe que tem o comportamento do formulário implementado, na qual os métodos realizam o controle e delegação de quem atende às requisições do usuário.

Neste exemplo agora proposto, surge a camada de modelo, também conhecida como camada de negócio. Ela é responsável por classes que fazem parte, de maneira natural, do problema a ser resolvido, como a leitura do consumo de energia, neste caso. Assim, a classe de modelo tem na visão os atributos que a compõe, e os métodos que capturam os eventos da visão controlam o fluxo de execução da aplicação.

Resolução 3.1: classe de negócio Leitura

```
namespace ConsumoEnergiaCondominio {
    class Leitura {
        public string Casa { get; set; }
        public double Consumo { get; set; }
        public double Desconto {
            get { return Consumo * 0.20; }
        }

        public Leitura(string casa, double consumo) {
            this.Casa = casa;
            this.Consumo = consumo;
        }

        public override bool Equals(Object l) {
            Leitura leitura = l as Leitura;
            if (litura  == null)
```

```
                return false;
            return (Casa.Equals(leitura.Casa));
        }

        public override int GetHashCode() {
            return new { Casa, Consumo}.GetHashCode();
        }
    }
}
```

A classe Leitura tem sua implementação dentro do namespace ConsumoEnergiaCondominio. O código da classe começa com a definição de suas propriedades Casa, Consumo e Desconto. Com exceção da propriedade Desconto, que é apenas de leitura (get), as outras duas são de escrita e leitura (set/get).

Note que, na definição de uma propriedade que simplesmente recebe ou retorna um dado, basta a especificação do nome do método. Entretanto, o desenvolvedor tem total liberdade para implementar o comportamento desses métodos, de acordo com a sua necessidade.

Um **campo de uma classe**, que também pode ser chamado de variável para objetos, normalmente tem seu escopo definido por modificadores de acesso como **privado** (private). Com esta visibilidade, esses campos não podem ser acessados externamente ao objeto a que pertence. A combinação de campos e métodos públicos que os acessam é conhecido como **propriedade**. Cabe salientar que o termo "campo de uma classe" é o usado na documentação da tecnologia da Microsoft. Assim, o termo "atributo de uma classe" é equivalente a este.

Uma **propriedade** é um membro que fornece um mecanismo flexível para ler, gravar ou calcular o valor de um **campo** particular. Propriedades podem ser usadas como se fossem membros de dados públicos, mas são métodos realmente especiais, chamados acessores. Isso permite que os dados sejam acessados com facilidade e ainda ajuda a promover a segurança e a flexibilidade dos métodos.

Para criar uma propriedade no Visual Studio, no editor de códigos digite prop e, em seguida, pressione a tecla Tab. Assim, a estrutura básica de uma

propriedade será fornecida. Caso se deseje personalizar o método `set`, o valor recebido pelo método estará em uma variável implícita chamada `value`.

Uma vez que a leitura por casa deve ocorrer apenas uma vez, é preciso garantir que não sejam informadas, de maneira equivocada, duas ou mais leituras. Como os objetos serão armazenados em uma coleção, há possibilidades de se verificar se determinado objeto encontra-se em uma seleção. Para que essa comparação seja realizada, cada objeto precisa ter um meio único de ser identificado. Isso pode ser realizado com a implementação dos métodos `Equals()` e `HashCode()`, como pode ser verificado, em destaque, na sequência.

Resolução 3.2: classe de negócio Leitura, com destaque para os métodos Equals() e HashCode()

```
namespace ConsumoEnergiaCondominio {
    class Leitura {
        // Implementação ocultada
        public override bool Equals(Object l) {
            Leitura leitura = l as Leitura;
            if (leitura == null)
                return false;
            return (Casa.Equals(leitura.Casa));
        }

        public override int GetHashCode() {
            return new { Casa, Consumo}.GetHashCode();
        }
    }
}
```

O capítulo 2 apresentou o método `Equals()` (seção 2.2) e seu objetivo de comparação de objetos, por meio de suas propriedades. Entretanto, tal exemplo trabalhou com strings e a comparação foi realizada com base nelas. Todavia, é comum a comparação entre objetos de classes implementadas pelo programador, como no caso a `Leitura`. Quando essa necessidade ocorre, é importante que o programador sobrescreva os métodos `Equals()` e `HashCode()`, como foi feito na classe da **Resolução 3.1** e nesta (represen-

tam a mesma classe).

Um **código hash** é um valor numérico que é usado para identificar um objeto durante o teste de igualdade. Ele pode também servir como um índice para um objeto em uma coleção. A implementação padrão do método `GetHashCode()` não garante um único valor de retorno para diferentes objetos. Dessa maneira, a implementação padrão desse método não deve ser utilizada como o identificador único de um objeto para propósitos de *hashing*.

A implementação do método `Equals()` para a classe `Leitura` realiza uma operação de *cast* para atribuir a uma variável do tipo `Leitura` o objeto que recebeu como `Object`. Caso o *cast* seja bem sucedido, verifica-se se o valor da variável recebida é ou não nulo. A identificação única para a classe `Leitura` está apenas na propriedade `Casa`, uma vez que não é permitida a leitura de consumo para duas casas. Como a propriedade `Casa` é uma `string`, o método retorna o resultado da comparação da casa do objeto atual, com o objeto recebido.

Para sobrescrever um método no Visual Studio, digite `override` e pressione a tecla de espaço. Todos os métodos que possam sofrer a sobrescrita aparecerão. Basta selecionar o desejado e a implementação padrão será exibida, sendo suficiente apenas personalizar o código de acordo com as necessidades.

Quando se estende uma classe, existe sempre a possibilidade de que métodos implementados nela possam (ou devam) ser sobrescritos, ou seja, ter seu comportamento redefinido na subclasse. A **sobrescrita** (`overriding`) é um recurso do paradigma orientado a objetos. O método sobrescrito pode, ainda, fazer uso do comportamento definido em sua superclasse. Para um método poder ser sobrescrito em sua definição, ele precisa ser marcado com a palavra **virtual** e, ao ser sobrescrito, deve ter a palavra **override**.

Em relação à implementação personalizada para o método `GetHashCode()`, optou-se pela criação de um objeto de tipo anônimo, com os valores das propriedades `Casa` e `Consumo`, para que fosse obtido o valor de hash desse novo objeto. Existem diversas técnicas para a sobrescrita desse método, sendo a utilizada neste exemplo uma delas.

Tipos anônimos fornecem uma maneira conveniente para encapsular um conjunto de propriedades somente leitura, em um único objeto, sem a neces-

sidade de definir explicitamente um tipo. O nome do tipo é gerado dinamicamente pelo compilador, que também infere o tipo de cada propriedade. Para a criação de tipos anônimos, faz-se uso do operador `new`, juntamente com um inicializador de objeto: `{ }`.

Terminada toda a explanação sobre a classe de modelo `Leitura` e dos conceitos relacionados a ela, há a seguir a implementação da classe que representa a camada de apresentação ou visão (o formulário).

Resolução 3.3: código completo da classe que representa o formulário

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;

namespace ConsumoEnergiaCondominio {
    public partial class FormConsumo : Form {
        private IList<Leitura> leituras =
            new BindingList<Leitura>();
        private BindingSource leituraSource =
            new BindingSource();

        public FormConsumo () {
            InitializeComponent();
            leituraSource.DataSource = leituras;
            dgvLeituras.DataSource = leituraSource;
        }

        private void BtnRegistrar_Click(object sender,
                EventArgs e) {
            RegistraConsumo(txtCasa.Text,
            Convert.ToDouble(txtConsumo.Text));
        }

        private void RegistraConsumo(string casa,
                double consumo) {
```

```
        Leitura leitura = new Leitura(casa, consumo);
        if (leituras.Contains(leitura)) {
            MessageBox.Show("A leitura para esta casa
                já foi registrada", "Alerta",
                MessageBoxButtons.OK,
                MessageBoxIcon.Warning);
        }
        else {
            this.leituras.Add(leitura);
            InicializaFormulario();
        }
    }

    private void InicializaFormulario() {
        txtCasa.Clear();
        txtConsumo.Clear();
        txtCasa.Focus();
    }

    private void BtnProcessar_Click(object sender,
            EventArgs e) {
        ProcessarLeituras(dgvLeituras);
    }

    private void ProcessarLeituras(DataGridView dgv) {
        DataGridViewCell cell =
            dgvLeituras.Rows[0].Cells[0];
        this.leituras.Add(new Leitura("Total", 0));
        for (int i = 0; i < 3; i++) {
            dgv.Rows[dgvLeituras.Rows.Count - 1]
                .Cells[i].Style.BackColor =
                Color.Blue;
            dgv.Rows[dgvLeituras.Rows.Count - 1]
                .Cells[i].Style.ForeColor =
                Color.Yellow;
            dgv.Rows[dgvLeituras.Rows.Count - 1]
                .Cells[i].Style.Font =
                new Font(cell.InheritedStyle.Font,
                FontStyle.Bold);
```

```
            }
            double totalConsumo = 0, totalDesconto = 0;
            foreach (var leitura in leituras) {
                totalConsumo += leitura.Consumo;
                totalDesconto += leitura.Desconto;
            }
            dgv[0, dgv.Rows.Count - 1].Value = "Total";
            dgv[1, dgv.Rows.Count - 1].Value =
                totalConsumo.ToString("N");
            dgv[2, dgv.Rows.Count - 1].Value =
                totalDesconto.ToString("N");
            lblResultado.Text =
                "Total consumo sem desconto: " +
                (totalConsumo - totalDesconto)
                .ToString("N");
        }
    }
}
```

Logo no início da classe que representa o formulário de interação com o usuário, existem dois campos (ou variáveis) definidos: `leituras` e `leituraSource`. O primeiro, `leituras`, representa uma coleção. Já o segundo, `leituraSource`, representa um `BindingSource`, um objeto que serve como fonte de dados para um controle visual, visto em detalhes na sequência.

Resolução 3.4: definição dos campos que são utilizados na classe

```
// Código omitido
namespace ConsumoEnergiaCondominio {

public partial class FormConsumo : Form {
    // Código omitido

    private IList<Leitura> leituras =
        new BindingList<Leitura>();
    private BindingSource leituraSource =
        new BindingSource();
```

```
    // Código omitido
    }
}
```

A coleção `leituras` tem como tipo a interface `IList`, que, por meio de `generics`, garante que todos os dados a serem armazenados nela sejam do tipo `Leitura`. Por `IList` ser uma interface, não é possível obter um objeto direto desse tipo. Entretanto, a classe `BindingList` implementa essa interface e ela é instanciada. Assim, tem-se um objeto instanciado por `BindingList` do tipo `IList`. Com isso, é possível que esse objeto possa ser atribuído a qualquer variável de `IList`.

A necessidade de criar e gerenciar grupos de objetos relacionados é uma característica comum nas aplicações. Para implementar essa necessidade, é possível fazer uso de matrizes (arrays) ou **coleções** (`collections`). O uso de matrizes é útil quando existe rigidez no tipo de dado, e o número de elementos é conhecido. Já o uso de coleções propicia maior flexibilidade, permitindo um aumento ou redução na quantidade de objetos armazenados, de acordo com a necessidade da aplicação. Algumas coleções permitem a atribuição de chaves para cada objeto armazenado, possibilitando uma busca mais eficiente. Outros garantem que objetos iguais não sejam armazenados de maneira repetida.

A `IList` é uma interface genérica que representa uma coleção de dados, na qual os itens podem ser acessados por meio de um índice. Existem diversas classes que a implementam e uma delas é a `BindingList`.

A `BindingList*` é uma classe que implementa a interface `IList`. Ela representa coleções que precisam oferecer suporte à vinculação de dados com controles visuais, como o `DataGridView`. Os controles visuais que recebem objetos dessa classe têm sua visão atualizada ao mesmo tempo em que isso ocorre na coleção.

O componente `BindingSource` atende a muitos propósitos. Primeiro, ele simplifica a ligação de controles em um formulário a um conjunto de dados, fornecendo gerenciamento concorrente, notificações de alteração e outros serviços entre controles `Windows Forms` e fontes de dados. Isso é feito pela atribuição do `BindingSource` à propriedade `DataSource` do con-

trole que fará uso dos dados. Diversos recursos são disponibilizados por esse componente por meio de suas propriedades, métodos e eventos.

Após a declaração das variáveis de objeto, verifica-se no código o construtor implementado para a classe. Esse construtor, criado automaticamente pelo Visual Studio, agora, além de inicializar toda a interface com o usuário por meio da chamada ao método `InitializeComponent()`, atribui ao `BindingSource` a coleção criada para receber os objetos relacionados às leituras realizadas no condomínio.

Além disso, o código para o construtor termina com a atribuição do `BindingSource` à propriedade `DataSource` do `DataGridView`. Após essa atribuição, com base nas propriedades da classe explicitada na declaração e instanciação da coleção, o `DataGridView` consegue identificar as colunas que o vão compor, dando a elas o nome das propriedades públicas da classe – neste caso, a classe `Leitura`. A seguir, apresento o construtor da classe.

Resolução 3.5: código do construtor da classe do formulário

```
public FormConsumo () {
    InitializeComponent();
    leituraSource.DataSource = leituras;
    dgvLeituras.DataSource = leituraSource;
}
```

Uma **interface** contém apenas assinaturas de métodos. É praticamente como um contrato entre a classe e o mundo externo. Uma classe, ao herdar uma interface, está comprometida a fornecer o comportamento publicado por esta, ou seja, escrever todos os métodos assinados. Devido a essa obrigação pela implementação, utiliza-se o termo "a classe X implementa a interface Y".

Polimorfismo, que significa "várias formas", é um termo designado a objetos de classes distintas, que, ao fazerem o uso de um mesmo método, podem reagir de uma maneira diferente. É um dos responsáveis pela extensão em uma aplicação OO, pois comportamentos definidos, mas não implementados nas classes mais genéricas, serão implementados nas novas subclasses.

Com a aplicação em execução, o usuário deverá informar o número da casa onde a leitura ocorreu e o valor consumido por esta. Após a informação

desses dois dados, o usuário precisa registrar a leitura, o que é feito quando o botão "Registrar" é pressionado. O método %BtnRegistrar_Click() é o responsável pelo processamento do evento Click desse botão. A implementação desse método delega a responsabilidade para o método RegistraConsumo(), que recebe como argumentos o número da casa (string) e o valor do consumo (double).

A implementação do método BtnRegistrar_Click() pode ser visualizada na sequência.

Resolução 3.6: método Click do botão Registrar

```
private void BtnRegistrar_Click(object sender,
        EventArgs e) {
    RegistraConsumo(txtCasa.Text,
    Convert.ToDouble(txtConsumo.Text));
}
```

O método RegistraConsumo() instancia a classe Leitura, enviando ao seu construtor o número da casa e o consumo recebidos pelo método. Com o objeto criado, é preciso garantir que não esteja sendo realizada uma nova leitura para a mesma casa, pois o sistema deve garantir que apenas uma leitura por casa seja registrada.

Uma vez que a classe Leitura tem implementados os métodos Equals() e GetHashCode(), é possível fazer uso de serviços (métodos) disponibilizados pela classe BindingList, como o método Contains(). Este verifica a existência do objeto enviado como argumento na coleção em que é realizada a chamada a ele. Caso a leitura já tenha sido registrada, o usuário é informado; caso contrário, o objeto de Leitura é adicionado à coleção, por meio do método Add(). Para garantir que o usuário registre uma nova leitura, desde o início, o método InicializaFormulario() é invocado.

Na sequência, trago a implementação do método RegistraConsumo().

Resolução 3.7: implementação do método RegistraConsumo()

```
private void RegistraConsumo(string casa, double consumo){
    Leitura leitura = new Leitura(casa, consumo);
    if (leituras.Contains(leitura)) {
        MessageBox.Show("A leitura para esta casa
            já foi registrada", "Alerta",
            MessageBoxButtons.OK,
            MessageBoxIcon.Warning);
    }
    else {
        this.leituras.Add(leitura);
        InicializaFormulario();
    }
}
```

O método `InicializaFormulario()` limpa os `TextBoxs`, para uma nova entrada, e coloca o foco da aplicação (cursor) no controle que recebe o número da casa. A seguir, apresento este método.

Resolução 3.8: código para o método InicializaFormulario()

```
private void InicializaFormulario() {
    txtCasa.Clear();
    txtConsumo.Clear();
    txtCasa.Focus();
}
```

Após o registro de leituras de todas as casas, é preciso realizar o processamento delas, buscando identificar o solicitado pelo enunciado. O método `BtnProcessar_Click()` é o responsável por isso. Seu comportamento delega ao `ProcessarLeituras()` a responsabilidade pelo processamento, enviando o `DataGridView` como argumento. Na sequência, apresento o código para este método.

Resolução 3.9: código para o método referente ao evento Click do botão BtnProcessar

```
private void BtnProcessar_Click(object sender,
        EventArgs e) {
```

```
    ProcessarLeituras(dgvLeituras);
}
```

Garanta que o processamento das leituras ocorra apenas caso existam leituras realizadas.

O método `ProcessarLeituras()` traz em sua implementação duas estruturas de repetição; foco deste capítulo. No método, as duas instruções iniciais referem-se à:

1) Obtenção de uma célula do `grid`, que servirá de base para configuração de fonte, na linha de totais;

2) Inserção de uma linha ao final do `grid`, para exibição dos totais.

Em seguida, o primeiro bloco de repetição ocorre.

A seguir, reapresento o método `ProcessarLeituras()`, tendo comentado o trecho deste bloco.

Resolução 3.10: método ProcessarLeituras()

```
private void ProcessarLeituras(DataGridView dgv) {
    DataGridViewCell cell = dgvLeituras.Rows[0].Cells[0];
    this.leituras.Add(new Leitura("Total", 0));

    // Início da estrutura de repetição for()
    for (int i = 0; i < 3; i++) {
        dgv.Rows[dgvLeituras.Rows.Count - 1]
            .Cells[i].Style.BackColor = Color.Blue;
        dgv.Rows[dgvLeituras.Rows.Count - 1]
            .Cells[i].Style.ForeColor = Color.Yellow;
        dgv.Rows[dgvLeituras.Rows.Count - 1]
            .Cells[i].Style.Font =
            new Font(cell.InheritedStyle.Font,
            FontStyle.Bold);
    }

    // Término da estrutura de repetição for()
```

```
    double totalConsumo = 0, totalDesconto = 0;

    // Início da estrutura de repetição foreach()
    foreach (var leitura in leituras) {
        totalConsumo += leitura.Consumo;
        totalDesconto += leitura.Desconto;
    }
    // Término da estrutura de repetição foreach()

    dgv[0, dgv.Rows.Count - 1].Value = "Total";
    dgv[1, dgv.Rows.Count - 1].Value =
        totalConsumo.ToString("N");
    dgv[2, dgv.Rows.Count - 1].Value =
        totalDesconto.ToString("N");

    lblResultado.Text =
        "Total consumo sem desconto: " +
        (totalConsumo - totalDesconto).
        ToString("N");
}
```

A estrutura de **repetição contada** é recomendada quando se sabe previamente por quantas vezes um determinado conjunto de instruções (ou um terminado processo) deve ser repetido. Essa quantidade de vezes pode ser um valor constante ou variável, de acordo com a necessidade do usuário ou do problema. A instrução `for()` é a implementação da estrutura de repetição contada para o C# e em diversas outras linguagens.

Aqui, é possível verificar que existem três instruções que estão dentro do bloco `for()`. Desta maneira, assim como nas estruturas condicionais, há a necessidade de delimitadores de bloco (chaves). Caso apenas uma instrução fosse executada de maneira repetitiva, as chaves seriam opcionais.

A instrução possui, em seus argumentos, três expressões componentes (separadas por ponto e vírgula). A primeira é conhecida como **inicializadora**, pois declara a variável (ou variáveis) de controle. A segunda **avalia a condição para repetição**, normalmente um valor inferior (ou superior) a outro, que limita a quantidade de execuções do bloco. Já a terceira traz o **processamento** que deve ser executado após cada iteração do bloco. Há situações em

que programadores utilizam essa instrução apenas com a segunda expressão, como, por exemplo, `for(;x!=null;)`. Entretanto, isso não é comum, pois existem estruturas próprias para esta situação.

As instruções dentro do bloco `for()` são responsáveis por alterar a formatação visual das três colunas do `DataGridView`, da linha que apresentará os totais – no caso deste problema, a última linha. Neste sentido, cada iteração realizará a configuração para cada célula da linha de totais. Por isso, a célula (`Cells`) é indicada por `i`, que vai de 0 (zero) até 2 (menor que 3).

Após o término dessa primeira estrutura de repetição, variáveis "acumuladoras" são inicializadas (`totalConsumo` e `totalDesconto`). Elas serão atualizadas com base em uma varredura realizada na coleção que armazena as leituras, com a instrução `foreach()`.

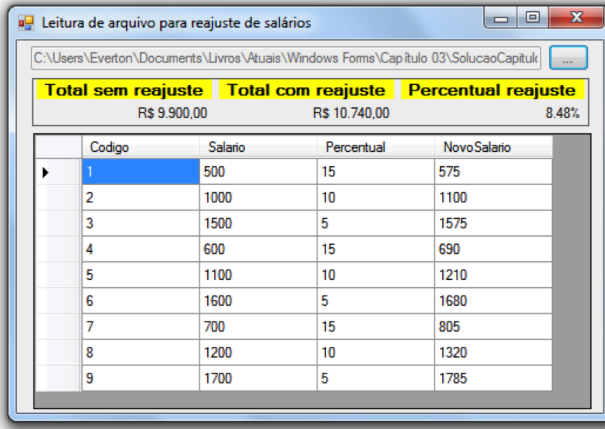
A estrutura `foreach` é responsável pela leitura de todos os elementos de uma coleção, associando cada um deles a cada iteração, a uma variável.

Finalizando o método, após o `foreach`, os valores obtidos nas variáveis "acumuladoras" são atribuídos à última linha do `grid`, devidamente formatados, assim como o valor para o `Label` ao lado do `Button`, que exibe o total consumido sem os descontos.

3.2 Leitura de arquivo texto para cálculo de reajuste em folha de pagamento

Para trabalharmos com novos recursos, desenvolva um aplicativo que deverá ler um arquivo texto, com valores delimitados por ponto e vírgula (`;`). Nele, os dados constantes referem-se ao código e ao salário de funcionários de uma empresa. Após a leitura desse arquivo, os dados lidos devem ser processados, objetivando a realização do cálculo para reajuste da folha de pagamento para essa empresa. Para o reajuste dos salários, que será por faixa, considere que:

- Para cada funcionário, o reajuste será de 15%, caso seu salário seja menor que R$ 1.000;
- Se o salário for maior ou igual a R$ 1.000, porém menor ou igual a R$ 1.500, seu reajuste será de 10%;

- Caso seja maior que R$ 1.500, o reajuste deverá ser de 5%.

Após a leitura do arquivo e identificação dos novos salários, a aplicação deverá exibir:

a) O valor total da folha de pagamento antes do reajuste;

b) O valor total da folha de pagamento após o reajuste;

c) O percentual de acréscimo à folha de pagamento, em relação à folha atual.

Na implementação proposta como resolução para este problema, uma nova técnica será trabalhada: o uso de arquivos textos como fonte de dados. Dois novos controles também serão apresentados: o `Panel` e o `TableLayoutPanel`. Esses controles são containers e possibilitam uma boa organização de controles visuais no desenho da janela da aplicação.

A única interação do usuário com a aplicação será a seleção do arquivo de texto, que possui as informações necessárias para o processamento desejado. Após a seleção correta, o arquivo deverá ser lido linha a linha, cada qual com os dados separados por um sinal de ponto e vírgula (`;`) e representa um objeto de uma classe de negócio. Com toda a leitura realizada, quando os salários de cada funcionário forem aplicados à regra para reajuste, os totais solicitados serão exibidos.

A figura 3.2 representa a interface desejada, que será implementada para o problema proposto nesta seção.

Leitura de arquivo para reajuste de salários

C:\Users\Everton\Documents\Livros\Atuais\Windows Forms\Capítulo 03\SolucaoCapitulo

Total sem reajuste	Total com reajuste	Percentual reajuste
R$ 9.900,00	R$ 10.740,00	8.48%

Codigo	Salario	Percentual	NovoSalario
1	500	15	575
2	1000	10	1100
3	1500	5	1575
4	600	15	690
5	1100	10	1210
6	1600	5	1680
7	700	15	805
8	1200	10	1320
9	1700	5	1785

Fig. 3.2: Interface para o problema de reajuste de folha de pagamento

Os dados apresentados na figura 3.2 foram lidos de um arquivo texto (`*.txt`) com os valores (código e salário atual) separados por ponto e vírgula. A figura 3.3 traz o conteúdo desse arquivo.

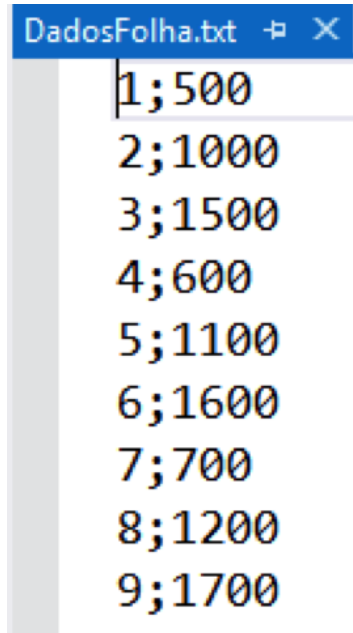



Fig. 3.3: Conteúdo de arquivo texto a ser utilizado na resolução

Crie um projeto `Windows Forms Application` na solução criada anteriormente, com um formulário específico para a implementação da janela da figura 3.2. Os controles que exibem os totais são: `Panel`, `TableLayoutPanel` e `Labels`. Acima desses totais existe um `TextBox` desabilitado (`Enabled=False`) e um `Button`. A exibição das leituras, abaixo dos totais, é realizada por meio de um `DataGridView`.

O controle `Panel` é usado para fornecer um agrupamento para outros controles e, normalmente, para subdividir um formulário por características de dados ou funcionalidades.

Como exemplo, em um formulário poderia existir dados relacionados às contas a pagar e às contas pagas, logo, poderiam existir dois `Panels`, um para cada tipo de conta. Esta técnica de agrupamento pode propiciar ao usuário uma visão lógica dos dados. Em tempo de desenho, quando se move um `Panel`, todos os controles inseridos nele são movidos em conjunto. Um `Panel` possui uma propriedade chamada `Controls` – todo container tem

essa propriedade –, que permite acessar todos os controles contidos nele em tempo de execução.

Já o controle `TableLayoutPanel` organiza seu conteúdo em uma tabela. Pelo fato de o layout ser realizado tanto em tempo de desenho quanto de execução, ele pode mudar dinamicamente, de acordo com a mudança do ambiente da aplicação. Esse controle oferece aos controles que ele contém a habilidade de ter seu tamanho alterado proporcionalmente.

Qualquer controle pode pertencer ao `TableLayoutPanel`, incluindo outros `TableLayoutPanels`. Isso permite uma criação de layouts sofisticados. Ele adiciona algumas propriedades aos controles contidos nele, sendo elas: `Cell`, `Column`, `Row`, `ColumnSpan` e `RowSpan`.

No exemplo anterior, a coleção dos dados informados pelo usuário estava declarada diretamente na classe do formulário que os solicitava. Uma boa prática é ter uma classe específica para isso, evitando uma coesão forte entre as camadas de apresentação, de negócio e de persistência. Dessa maneira, esse exemplo traz o uso de uma classe que aplica o conceito de repositório. Ela será responsável por ter a coleção dos dados informados pelo usuário, assim como os métodos para manutenção e acesso a eles. Na sequência, há a implementação dessa classe, chamada `RepositorioFuncionario`.

Resolução 3.11: classe responsável por manter e fornecer dados dos funcionários

```
namespace ReajusteDeFolhaDePagamento {
    public class RepositorioFuncionario {

    private IList<Funcionario> funcionarios =
        new BindingList<Funcionario>();

    public void Inserir(Funcionario funcionario){
        funcionarios.Add(funcionario);
    }

    public IList<Funcionario> ObterTodos() {
        return this.funcionarios;
    }
```

```
    }
}
```

A classe `RepositorioFuncionario` tem sua implementação dentro do `namespace ReajusteDeFolhaDePagamento`. Sua implementação tem, em seu início, a definição de um campo privado, responsável pela coleção de dados, sendo estes representados pela classe `Funcionario`. Inicialmente, a classe oferece dois métodos (serviços): um para a inclusão de um novo funcionário e outro que retorna todos os empregados registrados na coleção deste repositório. A seguir, apresento a classe `Funcionario`.

Resolução 3.12: classe de negócio para registro de dados de cada Funcionario

```
namespace ReajusteDeFolhaDePagamento {
    public class Funcionario {
        public int Codigo { get; set; }
        public double Salario { get; set; }
        public double Percentual {
            get {
                if (this.Salario < 1000) return 15;
                else if (this.Salario < 1500) return 10;
                else
                    return 5;
            }
        }
        public double NovoSalario {
            get {
                return (this.Salario * this.Percentual /
                    100)+this.Salario; }
        }
    }
}
```

A classe `Funcionario` traz 4 atributos. Dois destes permitem escrita e leitura, pois são os que receberão os dados lidos do arquivo de texto (`Codigo` e `Salario`), e os outros dois (`Percentual` e `NovoSalario`) são apenas de leitura, com base em processamento no valor do `Salario`. Note que o

processamento realizado no `get()` da propriedade `Percentual` refere-se à identificação do percentual de reajuste a ser aplicado, de acordo com o exposto no enunciado. Já a propriedade `NovoSalario` realiza apenas um cálculo para aplicar no salário atual o percentual identificado.

Uma vez que a leitura dos dados será realizada com base em um arquivo de texto, é preciso fornecer ao usuário da aplicação mecanismos para seleção desse arquivo. O C# oferece um componente chamado `OpenFileDialog`, que possibilita selecionar um arquivo ou mais. Essa seleção ocorre por meio da janela básica oferecida pelo Windows para abertura de arquivos. É uma janela comum em qualquer aplicação e pode ser verificada na figura a seguir:

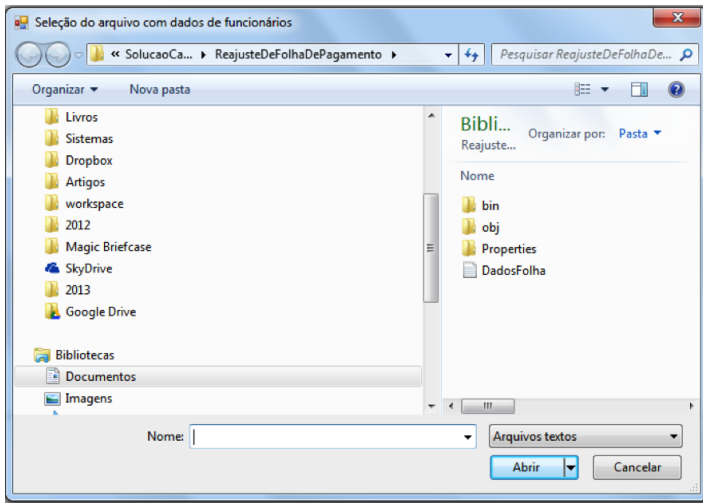



Fig. 3.4: Janela para seleção do arquivo de leitura

Para inserir o componente `OpenFileDialog` em seu formulário, para que possa ser usado, é preciso arrastá-lo para a área de desenho. Localize-o na categoria `Dialogs`, na `Toolbox`, e arraste-o. Ele será posicionado em uma área abaixo da de desenho, que não é visual. Essa área foi apresentada no capítulo 2 (seção 2.3). Altere o valor das propriedades para esse controle conforme mostrado na sequência.

- **Propriedade:** Name

- **Valor para atribuir:** ofdListaFuncionarios:

- **Propriedade:** DefaultExt
- **Valor para atribuir:** *.txt:

A propriedade `DefaultExt` define, dentre os tipos de arquivos apontados na propriedade `Filter`, qual é o padrão que será utilizado quando a janela de seleção de arquivos for aberta.

- **Propriedade:** Filter
- **Valor para atribuir:** Arquivos textos|*.txt:

A propriedade `Filter` define um conjunto de pares que informam os tipos de arquivos possíveis de serem selecionados na janela. Cada filtro é especificado por um nome que o descreve e sua extensão, separados por um *pipe* (`|`).

- **Propriedade:** Title
- **Valor para atribuir:** Seleção do arquivo com dados de funcionários

A propriedade `Title` define o título a ser atribuído para a janela a ser aberta por meio do componente.

Terminada toda a explanação sobre as classes de repositório e de negócio, assim como sobre o novo componente utilizado, trago na sequência a implementação da classe que representa a camada de apresentação (o formulário).

Resolução 3.13: classe que representa o formulário de interação com o usuário

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace ReajusteDeFolhaDePagamento {
```

```
public partial class FormLeituraArquivo : Form {
    private RepositorioFuncionario repositorio =
        new RepositorioFuncionario();
    private BindingSource leituraSource =
        new BindingSource();

    public FormLeituraArquivo() {
        InitializeComponent();
        leituraSource.DataSource =
            repositorio.ObterTodos();
        dgvLeitura.DataSource = leituraSource;
    }

    private void button1_Click(object sender,
            EventArgs e) {
        if (ofdListaFuncionarios.ShowDialog() ==
                DialogResult.OK) {
            txtArquivo.Text = ofdListaFuncionarios.
                FileName;
            ProcessarArquivo(txtArquivo.Text);
            if (repositorio.ObterTodos().Count > 0) {
                TotalizarValores(repositorio.
                    ObterTodos());
            }
        }
    }

    private void TotalizarValores(IList<Funcionario>
            dadosLidos) {
        double totalSemReajuste=0,totalComReajuste=0;
        foreach (var funcionario in dadosLidos) {
            totalSemReajuste += funcionario.Salario;
            totalComReajuste += funcionario.
                NovoSalario;
        }
        double percentualReajuste = (totalComReajuste
            - totalSemReajuste)*100/totalSemReajuste;
        lblTotalSemReajuste.Text = string.Format(
            "{0:c}", totalSemReajuste);
```

```
            lblTotalComReajuste.Text = string.Format(
                "{0:c}", totalComReajuste);
            lblPercentualDoReajuste.Text = string.Format(
                "{0:n}%", percentualReajuste);
        }

        private void ProcessarArquivo(string nomeArquivo){
            repositorio.ObterTodos().Clear();
            string linhaLida;
            var arquivo =
                new System.IO.StreamReader(@nomeArquivo);
            while ((linhaLida = arquivo.ReadLine()) !=
                    null) {
                var dadosLidos = linhaLida.Split(';');
                var funcionario = new Funcionario
                {
                    Codigo = Convert.ToInt32(
                        dadosLidos[0]),
                        Salario = Convert.ToDouble(
                        dadosLidos[1])
                };
                repositorio.Inserir(funcionario);
            }
            arquivo.Close();
        }
    }
}
```

Logo no início da classe que representa o formulário de interação com o usuário, existem dois campos (ou variáveis) definidos: `repositorio` e `leituraSource`. O primeiro representa a classe responsável pela manutenção dos dados para processamento e o segundo representa um `BindingSource`.

Ainda na classe que representa o formulário, após as propriedades verifica-se que, no construtor, o `BindingSource` recebe a coleção retornada pelo método `ObterTodos()` da classe de repositório. Note que, neste momento, nenhum dado foi ainda adicionado. Entretanto, com essa implementação, todo dado inserido será automaticamente exibido no `DataGridView`,

que tem em sua propriedade DataSource esse BindingSource. Esta ligação também pode ser verificada no construtor.

Com a aplicação em execução, o usuário deverá selecionar o arquivo que possui os dados necessários para processamento. Para isso, ele precisará clicar no botão que tem três pontos (...) e, então, buscar pelo arquivo e selecioná-lo. O método BtnSelecionarArquivo_Click() é o responsável pela interface que permitirá isso e também pelo início de processamento sobre os dados contidos nesse arquivo. A seguir, reapresento a implementação desse método.

Resolução 3.14: método Click do botão de seleção do arquivo a ser lido

```
private void BtnSelecionarArquivo_Click(object sender,
        EventArgs e) {
    if (ofdListaFuncionarios.ShowDialog() ==
            DialogResult.OK) {
        txtArquivo.Text = ofdListaFuncionarios.FileName;
        ProcessarArquivo(txtArquivo.Text);
        if (repositorio.ObterTodos().Count > 0) {
            TotalizarValores(repositorio.ObterTodos());
        }
    }
}
```

A primeira linha implementada nesse método é responsável pela execução do componente OpenFileDialog – pela chamada ao método ShowDialog() – e também pela verificação da seleção do arquivo desejado. Essa verificação dá-se pela comparação do valor retornado pelo método com a enumeração DialogResult.OK. Em caso verdadeiro, isso garante que o usuário selecionou um arquivo e clicou no botão OK da janela responsável pela localização deles. Ainda, em caso de sucesso, o arquivo selecionado – armazenado na propriedade FileName do controle – é atribuído ao TextBox, apenas por caráter informativo ao usuário.

Em seguida, o arquivo é enviado para processamento pelo método ProcessarArquivo(). Terminado o processamento, verifica-se se os dados foram inseridos no repositório e, em caso positivo, eles são processados

em busca dos totalizadores desejados.

O método `ProcessarArquivo()` é exibido na sequência e o `TotalizarValores()` na **Resolução 3.16**.

Resolução 3.15: método responsável pelo processamento do arquivo lido

```
private void ProcessarArquivo(string nomeArquivo) {
    repositorio.ObterTodos().Clear();
    string linhaLida;
    var arquivo = new System.IO.StreamReader(
        @nomeArquivo);
    while ((linhaLida = arquivo.ReadLine()) != null) {
        var dadosLidos = linhaLida.Split(';');
        var funcionario = new Funcionario {
            Codigo = Convert.ToInt32(dadosLidos[0]),
                Salario = Convert.ToDouble(
                dadosLidos[1])
        };
        repositorio.Inserir(funcionario);
    }
    arquivo.Close();
}
```

A primeira linha do método `ProcessarArquivo()` realiza uma limpeza na coleção do repositório. Essa limpeza é realizada pelo método `Clear()`, definido na interface `IList`. Na sequência, inicia-se o procedimento para leitura do arquivo selecionado.

O primeiro passo para isso é abrir o arquivo, o que é feito pela classe `StreamReader`. Após obter a referência para o arquivo, uma estrutura de repetição `while()` é iniciada, realizando a leitura do arquivo, uma linha por vez, até que seu final seja encontrado. A leitura dá-se pelo método `ReadLine()`.

Como em cada linha do arquivo existem dois dados, que estão separados por um ponto e vírgula (`;`), é preciso que eles sejam armazenados de maneira isolada, para então serem processados. Essa separação é realizada pelo método `Split()`, definido na classe `String`. O retorno desse método é

uma matriz com colunas que representam cada dado encontrado e separado na linha lida. Com os dados lidos, objetos da classe `Funcionario` são instanciados e, por meio do recurso de inicialização de objetos, os valores são atribuídos às propriedades do novo objeto, que, então, é inserido no repositório.

A classe `System.IO.StreamReader` destina-se à leitura de caracteres em uma codificação específica, como UTF-8 ou ANSI, por exemplo. Seu uso é destinado para a leitura de linhas em um arquivo de texto.

A estrutura `while` é responsável pela execução de um conjunto de instruções que estejam dentro dos delimitadores de bloco oferecidos por ela. Esta função (`while()`) recebe um argumento que é uma expressão lógica, que garantirá a execução desse conjunto de instruções, caso avaliada como verdadeira. Se apenas uma instrução compuser a estrutura `while`, o uso de delimitadores será opcional.

Resolução 3.16: método responsável pela totalização dos valores lidos

```
private void TotalizarValores(
        IList<Funcionario> dadosLidos) {
    double totalSemReajuste = 0, totalComReajuste = 0;
    foreach (var funcionario in dadosLidos) {
        totalSemReajuste += funcionario.Salario;
        totalComReajuste += funcionario.NovoSalario;
    }
    double percentualReajuste = (totalComReajuste -
        totalSemReajuste)*100/totalSemReajuste;
    lblTotalSemReajuste.Text = string.Format("{0:c}",
        totalSemReajuste);
    lblTotalComReajuste.Text = string.Format("{0:c}",
        totalComReajuste);
    lblPercentualDoReajuste.Text =
        string.Format("{0:n}%", percentualReajuste);
}
```

Este método primeiramente inicializa as variáveis acumuladoras, que terão os totais desejados. Após isso, uma varredura nos valores recebidos pelo

método é realizada, para totalizar os valores a serem exibidos. Após a obtenção dos totais, o percentual de reajuste é calculado e, então, os valores são atribuídos aos respectivos `Labels` da janela.

3.3 Informação de dados para geração de arquivo texto

Como terceiro (e último) problema proposto para este capítulo, desenvolva um aplicativo que permita a inserção de dados relacionados aos funcionários de uma empresa. Os dados solicitados são apenas nome e salário. Para esse problema, propõe-se novamente o uso de arquivos textos, porém agora como destino dos dados informados na aplicação. Nenhum controle visual novo será apresentado.

A quantidade de funcionários que deverão ter seus dados informados deve ser passada previamente pelo usuário, e os dados de todos os empregados deverão ser armazenados em um arquivo texto, delimitados por ponto e vírgula (`;`). Para esta exportação de dados, ele selecionará a pasta onde o arquivo será criado, assim como informará um nome para ele.

A figura 3.5 representa a interface desejada, que será implementada para o problema proposto nesta seção.

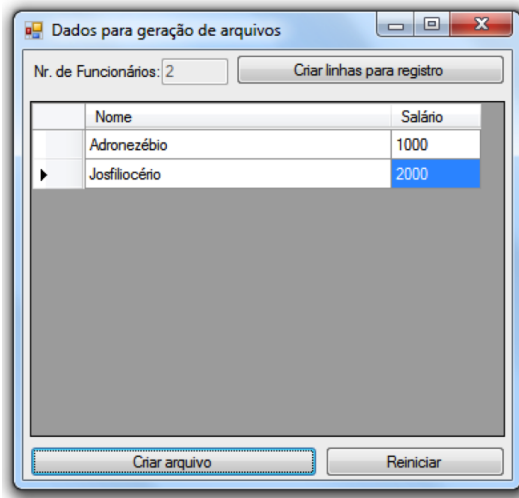



Fig. 3.5: Interface para o problema de criação de arquivo texto

Crie um projeto `Windows Forms Application` na solução criada anteriormente, com um formulário específico para a implementação da janela, de acordo com a proposta de interface apresentada na figura 3.5. Todos os controles usados já foram apresentados em exemplos anteriores. Configure os botões `Criar arquivo` e `Reiniciar` com a propriedade `Enabled`, com o valor `False`.

No exemplo anterior, um arquivo de texto foi utilizado para fornecer dados para a realização de processamentos relacionados a uma folha de pagamento. Agora, um arquivo de texto será gerado a partir de dados informados na aplicação. Sendo assim, é preciso fornecer ao seu usuário mecanismos para geração desse arquivo. Para isso, será usado um componente chamado `SaveFileDialog`, que possibilita a seleção de uma pasta de destino para o arquivo a ser gerado, assim como nome a ser dado a ele. Esse processo ocorre por meio da janela básica oferecida pelo Windows para gravação de arquivos, retratada na figura a seguir.

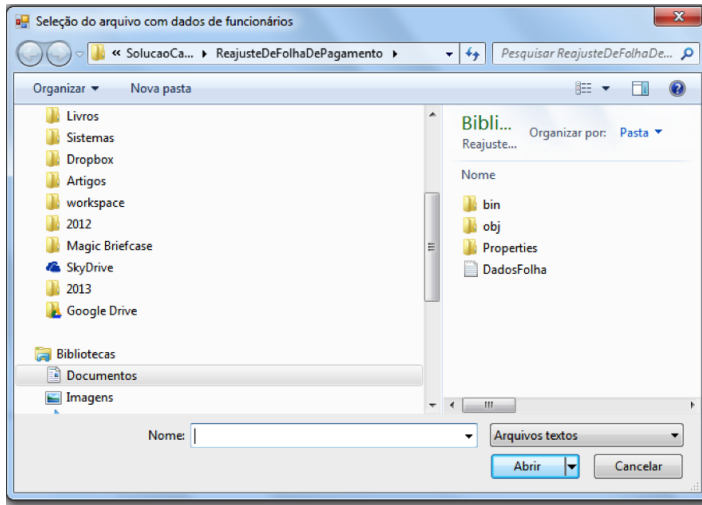



Fig. 3.6: Janela para seleção de pasta de destino para arquivo a ser gerado

O procedimento para a inserção do componente `SaveFileDialog` é semelhante ao usado para inserir o `OpenFileDialog`. Localize-o na categoria `Dialogs`, na `Toolbox`, e arraste-o. Ele será posicionado em uma área abaixo da área de desenho, que não é visual. Altere o valor das propriedades para esse controle, conforme apresentado na sequência.

Propriedades para o SaveFileDialog

- **Propriedade:** Name
- **Valor para atribuir:** sfdGravarArquivo:

- **Propriedade:** DefaultExt
- **Valor para atribuir:** *.txt

- **Propriedade:** Filter
- **Valor para atribuir:** Arquivos textos|*.txt

- **Propriedade:** Title
- **Valor para atribuir:** Dados para geração de arquivo

Terminada a explanação sobre o problema e sua interface de resolução, trago na sequência a implementação da classe que representa os métodos capturados pela camada de apresentação (o formulário).

Resolução 3.17: classe que representa o formulário de interação com o usuário

```
using System;
using System.IO;
using System.Windows.Forms;

namespace PreparacaoDeDadosParaGravacao {
    public partial class FormGeracaoArquivoTexto: Form {
        public FormGeracaoArquivoTexto () {
            InitializeComponent();
            dgvFuncionarios.ColumnCount = 2;
            dgvFuncionarios.Columns[0].HeaderText =
                "Nome";
            dgvFuncionarios.Columns[0].Width = 230;
            dgvFuncionarios.Columns[1].HeaderText =
                "Salário";
                dgvFuncionarios.Columns[1].Width = 67;
        }

        private void btnCriarLinhas_Click(object sender,
                EventArgs e) {
            int numeroFuncionarios = Convert.ToInt16(
                txtNumeroFuncionarios.Text);
            if (numeroFuncionarios < 1)
                numeroFuncionarios = 1;
            int i = 0;

            do {
                var linhaTabela = new DataGridViewRow();
                linhaTabela.Cells.Add(new
```

```
                DataGridViewTextBoxCell {
                    Value = string.Empty });
            linhaTabela.Cells.Add(new
                DataGridViewTextBoxCell { Value = 0 });
            dgvFuncionarios.Rows.Add(linhaTabela);
        } while (++i < numeroFuncionarios);

        txtNumeroFuncionarios.Enabled = false;
        btnCriarArquivo.Enabled = true;
        btnReiniciar.Enabled = true;
        btnCriarLinhas.Enabled = false;
    }

    private void btnReiniciar_Click(object sender,
            EventArgs e) {
        dgvFuncionarios.Rows.Clear();
        txtNumeroFuncionarios.Text = string.Empty;
        txtNumeroFuncionarios.Enabled = true;
        btnCriarArquivo.Enabled = false;
        btnReiniciar.Enabled = false;
        btnCriarLinhas.Enabled = true;
    }

    private void btnCriarArquivo_Click(object sender,
            EventArgs e) {
        if (!ValidaDados()) {
            MessageBox.Show( "Os dados possuem
                problemas. Verifique se não deixou
                nenhum nome em branco ou se existe um
                valor correto para os salários de cada
                um");
        }
        else if (sfdGravarArquivo.ShowDialog() ==
                DialogResult.OK) {
            GerarArquivo();
            MessageBox.Show("Arquivo gerado com
                sucesso");
        }
    }
```

```
        private void GerarArquivo() {
            StreamWriter wr = new StreamWriter(
                sfdGravarArquivo.FileName, true);
            for (int j = 0;
                    j < dgvFuncionarios.Rows.Count;
                    j++) {
                wr.WriteLine(dgvFuncionarios.Rows[j].
                    Cells[0].Value.ToString() + ";" +
                    dgvFuncionarios.Rows[j]. Cells[1].
                    Value.ToString());
            }
            wr.Close();
        }

        private bool ValidaDados() {
            int i = 0;
            bool dadosValidados = true;
            double stringToDouble;
            do {
                if (string.IsNullOrWhiteSpace(
                    dgvFuncionarios.Rows[i].Cells[0].Value.
                        ToString()))
                    dadosValidados = false;
                if (!Double.TryParse(dgvFuncionarios.
                        Rows[i].Cells[1].Value.ToString(),
                        out stringToDouble))
                    dadosValidados = false;
            } while (++i < dgvFuncionarios.Rows.Count);
            return dadosValidados;
        }
    }
}
```

Ao contrário das soluções anteriores deste capítulo, nessa classe que representa a janela de interação não existe a definição de um repositório e tampouco de um `BindingSource`. Para esse exemplo, a validação e uso dos dados para geração do arquivo ocorrerá diretamente sobre os dados informados no `DataGridView`.

Dessa maneira, verifica-se logo no início da classe o construtor, que tem a implementação padrão de todo formulário, chamando o método `InitializeComponents()`. Este definido no arquivo que controla as configurações visuais dos componentes inseridos no formulário e, em seguida, realiza configurações no `DataGridView`. Cabe lembrar que essas configurações poderiam também ter sido realizadas visualmente no controle.

Com a aplicação em execução, o usuário deverá informar a quantidade de funcionários que terão seus dados informados e clicar no botão `Criar linhas para registro`. O método que captura o evento `Click` para esse botão está representado na sequência.

Resolução 3.18: método que captura o evento clique do botão de criação de linhas

```
private void btnCriarLinhas_Click(object sender,
        EventArgs e) {
    int numeroFuncionarios = Convert.ToInt16(
        txtNumeroFuncionarios.Text);
    if (numeroFuncionarios < 1)
        numeroFuncionarios = 1;
    int i = 0;

    do {
        var linhaTabela = new DataGridViewRow();
        linhaTabela.Cells.Add(new DataGridViewTextBoxCell
            { Value = string.Empty });
        linhaTabela.Cells.Add(new DataGridViewTextBoxCell
        { Value = 0 });
        dgvFuncionarios.Rows.Add(linhaTabela);
    } while (++i < numeroFuncionarios);
    txtNumeroFuncionarios.Enabled = false;
    btnCriarArquivo.Enabled = true;
    btnReiniciar.Enabled = true;
    btnCriarLinhas.Enabled = false;
}
```

As primeiras instruções deste método são referentes à conversão do valor digitado no `TextBox` para inteiro e garantir que, ao menos, um funcionário

tenha seus dados informados. Em seguida, uma variável contadora que será usada na estrutura de repetição é declarada.

Após esses procedimentos, a estrutura de repetição `do...while()` é iniciada. Dentro do bloco, para cada funcionário que terá dados lidos é criada uma nova linha (classe `DataGridViewRow`). Depois, para a linha criada, são adicionadas células. Como uma célula pode ser ocupada por diversos tipos de controles, nesse exemplo foi utilizada a classe `DataGridViewTextBoxCell`, que insere na célula um `TextBox`.

Durante a instanciação da classe, já é inserido o valor `default` para ela. Tendo as duas células sido criadas, a linha é adicionada ao `DataGridView`, por meio do método `Add()` da propriedade `Rows`. Após a estrutura de repetição, o `TextBox` que recebe a quantidade de funcionários é desabilitado, e os botões para `Criar Arquivo` e `Reiniciar o processo`, habilitados. Assim, o botão `Criar linhas para registro` é também desabilitado.

A estrutura `do...while()` é responsável pela execução de um conjunto de instruções que estejam dentro dos delimitadores de bloco oferecidos por ela. Como a condição para continuidade ou não do laço está no final, as instruções dentro dela serão executadas ao menos uma única vez. Caso apenas uma instrução componha a estrutura `do...while()`, o uso de delimitadores é opcional.

Com a interface liberada para o usuário informar os dados, após estes terem sido informados, ele precisa iniciar o processo para geração do arquivo de texto. Para isso, ele precisará clicar no botão `Criar arquivo`, que tem o evento `Click` capturado pelo método apresentado a seguir.

Resolução 3.19: método que captura o evento clique no botão que dispara a criação do arquivo

```
private void btnCriarArquivo_Click(object sender,
        EventArgs e) {
    if (!ValidaDados()) {
        MessageBox.Show( "Os dados possuem
            problemas. Verifique se não deixou
            nenhum nome em branco ou se existe um
            valor correto para os salários de cada
            um");
```

```
    }
    else if (sfdGravarArquivo.ShowDialog() ==
            DialogResult.OK) {
        GerarArquivo();
        MessageBox.Show("Arquivo gerado com
            sucesso");
    }
}
```

Para a geração do arquivo texto com os dados informados pelo usuário, o primeiro passo é a realização de uma validação. Esse procedimento é realizado pela primeira instrução do método apresentado aqui, o `ValidaDados()`, invocado como argumento do `if()`. Essa validação tem instruções que serão executadas quando o método de validação retornar um valor `false`, ou seja, quando ela não for bem-sucedida. A seguir, trago o método `ValidaDados()`.

Resolução 3.20: método que realiza a validação dos dados informados no DataGridView

```
private bool ValidaDados() {
    int i = 0;
    bool dadosValidados = true;
    double stringToDouble;
    do {
        if (string.IsNullOrWhiteSpace(
                dgvFuncionarios.Rows[i].Cells[0].Value.
                ToString()))
            dadosValidados = false;
        if (!Double.TryParse(dgvFuncionarios.
                Rows[i].Cells[1].Value.ToString(),
                out stringToDouble))
            dadosValidados = false;
    } while (++i < dgvFuncionarios.Rows.Count);
    return dadosValidados;
}
```

O processo de validação dos dados informados é relativamente simples. São apenas duas as situações verificadas:

1) Se a coluna referente ao nome do funcionário possui um valor válido (para isso é utilizado o método estático `IsNullOrWhiteSpace()` da classe `string`);

2) Se a coluna referente ao valor do salário possui um valor numérico correto (esta verificação é realizada pelo método estático `TryParse()` da classe `Double`).

O método `string.IsNullOrWhiteSpace()`, como o próprio nome diz, verifica se a `string` enviada como argumento é nula, ou se seu conteúdo é formado apenas por espaços em branco. Seu uso é indicado para validação de strings. Outro método similar é `string.IsNullOrEmpty()` ou ainda a propriedade `string.Empty`.

O método `Double.TryParse()` tenta converter um dado `string` para um `double`. Em caso de sucesso, ele retorna `true` e armazena o valor em seu argumento de saída; caso contrário, o método retorna `false` e o valor de saída recebe o (zero).

Continuando a explanação da **Resolução 3.19**, caso os dados informados estejam corretos e o processo de validação retorne um valor `true`, a instrução `else if()` é executada, invocando o método `ShowDialog()` do objeto `sdfGravarArquivo`. Esse método exibirá ao usuário a janela responsável para escolha do local onde o arquivo será gravado, assim como a informação do seu nome.

Caso o usuário realize de maneira correta essa operação, as instruções dentro do bloco do `else if()` serão executadas e o método `GerarArquivo()` será invocado para a criação do arquivo texto com os dados informados pelo usuário. Na sequência, apresento esse método.

Resolução 3.21: método responsável pela geração do arquivo com os dados informados no DataGridView

```
private void GerarArquivo() {
    StreamWriter wr = new StreamWriter(
        sfdGravarArquivo.FileName, true);
    for (int j = 0;
            j < dgvFuncionarios.Rows.Count;
```

```
            j++) {
        wr.WriteLine(dgvFuncionarios.Rows[j].
            Cells[0].Value.ToString() + ";" +
            dgvFuncionarios.Rows[j]. Cells[1].
            Value.ToString());
    }
    wr.Close();
}
```

O processo de geração do arquivo é relativamente simples. Obtém-se uma referência para o arquivo que será criado, por meio da classe `StreamWriter`, e fazendo uso do método `WriteLine()` do objeto dessa classe, os dados são escritos. Ao final, o objeto `StreamWriter` é fechado, gravando, assim, o arquivo.

A classe `System.IO.StreamWriter` destina-se à escrita de caracteres em uma codificação específica, como UTF-8 ou ANSI, por exemplo. Seu uso é destinado para a escrita de linhas em um arquivo de texto.

O terceiro botão disponibilizado na janela possibilita ao usuário a reinicialização do processo como um todo. Apresento, a seguir, o código do evento `Click` do botão `Reiniciar`.

Resolução 3.22: método que captura o evento de clique no botão para reiniciar o processo

```
private void btnReiniciar_Click(object sender,
        EventArgs e) {
    dgvFuncionarios.Rows.Clear();
    txtNumeroFuncionarios.Text = string.Empty;
    txtNumeroFuncionarios.Enabled = true;
    btnCriarArquivo.Enabled = false;
    btnReiniciar.Enabled = false;
    btnCriarLinhas.Enabled = true;
}
```

O processo para reiniciar a informação dos dados se dá, como pode ser verificado pelas instruções, ao limpar as linhas do `DataGridView` e do `TextBox` referente à quantidade de funcionários que serão informados. O estado desse `TextBox` é também habilitado. Em seguida, os botões `Criar`

`arquivo` e `Reiniciar` são desabilitados, e o `Criar linhas` habilitado. Com isso, a aplicação retorna ao estado inicial, como se tivesse sido executada pela primeira vez.

Em qualquer uma das estruturas de repetição mostradas anteriormente, é possível usar a instrução `break` para cancelar a repetição e sair do laço, ou ainda forçar a próxima iteração com a instrução `continue`. Quando se opta pelo uso da instrução `break`, a estrutura é interrompida, independentemente da condição imposta para término ter ocorrido ou não. Já quando se faz uso da instrução `continue`, a condição da estrutura é reavaliada e reiniciada (ou concluída), não sendo executadas na iteração as instruções que estiverem após a instrução `continue`.

3.4 Conclusão

Este capítulo apresentou informações essenciais sobre estruturas de repetição, que são muito utilizadas na programação. É preciso entender as quatro estruturas apresentadas, pois existem situações para que cada uma seja usada.

Para a aplicação dessas novas estruturas, foram usados novos recursos e novas técnicas. Dentre elas, destacam-se a escrita e leitura de arquivo texto, uma vez que esse processo é comumente utilizado em diversas aplicações comerciais.

Com este capítulo, conclui-se a etapa básica para conhecimento de qualquer linguagem. Com os três tipos de estrutura apresentados – sequencial, seleção/condicional e repetição –, é possível a resolução da maioria dos problemas que possam aparecer. É claro que particularidades sempre ocorrerão, o que o levará a conhecer recursos específicos da linguagem escolhida. Alguns desses recursos são referentes a classes que dependem uma da outra ou um conjunto de determinado tipo de dado/objeto, que são apresentados a partir do próximo capítulo.

Capítulo 4

Vários projetos em uma solução: aplicando alguns conceitos relacionados ao MVC e à Orientação a Objetos

Nos três capítulos anteriores, a Orientação a Objetos foi apresentada e aplicada. Entretanto, as explicações mantiveram-se apenas nos conceitos, e as aplicações basearam-se em algumas técnicas e usos de recursos já disponibilizados pela linguagem e ferramentas. Também nos capítulos anteriores, cada solução possuía apenas um projeto.

Este capítulo se dedicará à criação de uma solução com diversos projetos, buscando aplicar conceitos de MVC e da OO. Os exemplos farão uso tam-

bém, de uma maneira mais real, de coleções, onde elas serão utilizadas para a aplicação do conceito de associações entre objetos. Recursos visuais não serão trabalhados neste capítulo, mas procure aplicar tudo que foi apresentado nos anteriores, para enriquecer suas aplicações.

4.1 Introdução dos conceitos

Os conceitos apresentados de maneira teórica neste capítulo não se prendem a nenhuma linguagem de programação. São pontos importantes e necessários que balizam o modelo orientado a objetos e de que todo programador deve ter conhecimento, mesmo que seja apenas o básico. Alguns deles já foram apresentados durante as resoluções propostas nos capítulos anteriores, mas, por objetivos didáticos, se concentram novamente aqui. Na sequência, apresento-os com mais detalhes.

4.1.1 Objeto

Um objeto pode ser interpretado como qualquer estrutura modular que faça parte de algo, seja um carro, uma pessoa ou um acontecimento etc. Tendo uma janela como objeto, pode-se afirmar que ela faz parte de uma casa, de um carro ou ainda de uma aplicação (software). Tanto a casa como o carro ou a aplicação podem também ser vistos como objetos, compondo objetos maiores, ou ainda, se **associando** a objetos maiores.

Em uma visão mais abstrata, um objeto pode ser uma expressão matemática ou uma equação, nas quais os operandos e operadores, além do resultado, podem também ser vistos como objetos. É importante ter conhecimento de que cada objeto possui **propriedades**, **comportamentos** e **métodos**, que o identificam e diferenciam dentre outros objetos semelhantes.

Propriedades

Uma propriedade pode ser conceituada como a interface de um objeto com o mundo externo. Esse mundo externo pode ser visto como outros objetos relacionando-se entre si, que também podem ser chamados de objetos clientes do serviço oferecido. A interface de um objeto é aquilo que o usuário (cliente) do objeto (por exemplo, outro objeto) visualiza e ao qual tem acesso.

Retornando ao exemplo do carro como objeto, a **Cor** do carro pode ser visualizada como uma propriedade. Com base nesse exemplo, identifica-se que as propriedades caracterizam um objeto, ou ainda, o estado dele. Dois carros podem ser diferenciados entre si por meio de suas propriedades. Eles podem ter muitas características em comum, mas uma o diferenciará: o número do **chassi**. Assim, identifica-se também a **identidade** do objeto, que é uma propriedade também.

Comportamentos

Em relação aos comportamentos possíveis (ou esperados) por um objeto, é possível defini-los como reações de um objeto em decorrência de alguma interação com ele.

Tendo um objeto mais comum com o objetivo deste livro, pegaremos como exemplo uma janela de uma aplicação. Nesta há um botão localizado na barra de título, que tem a responsabilidade de fechá-la. A ocorrência dessa responsabilidade é uma reação a um **evento** sofrido, que seria o **clique** nesse botão.

Desta maneira, pode-se afirmar que o comportamento de um objeto está relacionado às ações (em decorrência a reações) por ele executadas, normalmente relacionadas à interação entre objetos, ou ainda com o próprio usuário da aplicação.

Métodos

Como comentado na explanação sobre comportamentos, para cada evento ocorre uma reação e estes dois itens, juntos com os métodos, caracterizam a realização dos comportamentos. Os métodos, na realidade, realizam a reação dos eventos capturados pelos objetos.

No exemplo sobre o fechamento de uma janela de aplicação, a ação de fechar a janela precisa ocorrer, pois ela não se fecha sem uma instrução que realize este processo. Sempre que ocorre um evento, este **delega** a um método a responsabilidade de executar os procedimentos necessários para a realização do evento.

Um método é composto por um conjunto de instruções e é por meio deles que ocorre a interação entre objetos, o que em algumas literaturas é também

conhecido como*troca de mensagens entre objetos*. Ainda, de maneira simplificada, é possível dizer que um método representa as ações tomadas pelos objetos, ou seja, os serviços que ele disponibiliza e executa.

4.1.2 Classe

Na explicação sobre objetos foi dado **pessoa** como exemplo. Já sobre propriedades, o exemplo dado (cor dos carros) apresentou a situação de mais de um objeto de um mesmo tipo, precisando inclusive de alguma propriedade que identificasse (ou distinguisse) um objeto de outro. Com isso, pode-se identificar que os objetos são vistos como conjuntos, um conjunto de **pessoas** ou um conjunto de **carros**, por exemplo.

Quando foram apresentados comportamentos e métodos, foi comentado que existe a necessidade de um conjunto de instruções oferecidas e realizadas por cada objeto. Entretanto, não seria prático que cada objeto tivesse essas instruções implementadas nele? Inclusive pelo fato de que se espera que o comportamento (ou propriedades) de cada objeto, de um mesmo conjunto, seja o mesmo. Por exemplo, todos os carros possuem uma cor (propriedade) e sofrem processos de frenagem e aceleração (comportamento).

Para a implementação destes métodos e propriedades comuns, que serão utilizados por um conjunto de objetos, usa-se o conceito de classes. Uma classe pode ser vista como uma categorização de objetos comuns, uma **categoria** que representa um conjunto de **objetos**. Desta maneira, toda a codificação (implementação) para propriedades e métodos se dá na classe, e os objetos que representam as classes trazem em si a possibilidade de utilizá-los.

4.1.3 Abstração

A abstração não é um termo exclusivo da área de informática, é algo que dia a dia todos vivenciam ou realizam. Um exemplo, já fazendo uso da leitura de livros, pode ser uma situação em que várias pessoas leem uma narrativa de um romance no qual o personagem descreve uma paisagem.

Certamente cada leitor visualizará em sua mente uma paisagem totalmente diferente do que outros visualizam, ou, ao menos, com particularidades que a diferenciam. Este resultado é a capacidade de abstração que cada

indivíduo possui: que é a "materialização" de conceitos, ideias ou pensamentos. Levando o conceito de abstração para a área computacional, pode-se afirmar que ela é considerada como a habilidade de modelar características do mundo real, de um denominado problema em questão, que o programador busca resolver.

4.2 Implementando os conceitos apresentados

Com os conceitos básicos apresentados e com vista para a aplicação deles, o desenvolvimento de um pequeno exemplo será iniciado. Para esta atividade, o primeiro conceito a ser aplicado é o de abstração. Para que essa abstração possa ocorrer, é preciso um estudo sobre o problema para que, com a sua compreensão, um modelo possa ser obtido. Na sequência, o enunciado que apresenta o problema:

Supondo que uma empresa tenha a necessidade de uma aplicação que possibilite o registro de compras realizadas por ela, crie um modelo para o formulário que representará essa aplicação.

Com base no proposto e uma análise realizada, abstrai-se que o formulário para o modelo precisa contemplar:

- Os dados relacionados ao fornecedor dos produtos, como nome e CNPJ;
- Em relação aos dados pertencentes ao processo de registro, o formulário precisa das datas de emissão da nota pelo fornecedor e a data em que o produto tem a entrada registrada na empresa, além do número da nota fiscal;
- Para os produtos adquiridos, é preciso informar a descrição, o preço de custo e a quantidade comprada para cada um, além do código que o identifica.

Com a análise e a abstração sobre o enunciado realizadas e as propriedades básicas identificadas, é possível constatar que existem dois grupos de informações: um que se refere à nota de entrada (número, fornecedor e datas) e outro que se refere aos produtos adquiridos na compra. Desta maneira,

duas categorias de objetos são identificadas, ou seja, duas classes. Essa categorização é necessária, de maneira bem básica, pelo fato de que um processo de aquisição (a compra) relaciona-se a vários produtos. Assim, para cada objeto que representa uma nota de entrada, poderão existir vários objetos que representam os produtos.

A implementação inicial da classe que representa os dados da nota pode ser visualizada na sequência.

Resolução 4.1: classe que representa a Nota de Entrada (Versão 1)

```
using System;
namespace ModelProject {
    class NotaEntrada {
        public string NumeroNota { get; set; }
        public string NomeFornecedor { get; set; }
        public string CNPJ { get; set; }
        public DateTime DataEmissao { get; set; }
        public DateTime DataEntrada { get; set; }
    }
}
```

Para quem já tem um bom conhecimento em Orientação a Objetos (e também em banco de dados), certamente verificou que o modelo não está normalizado e tampouco está em uma granularidade correta para o paradigma de OO. Para um modelo e implementação corretos, seria necessário um particionamento da classe `NotaEntrada` em duas, separando os dados pertencentes ao fornecedor para uma classe específica (`Fornecedor`) e mantendo na classe original apenas os dados relacionados ao processo de compra.

A classe inicial que representa cada produto adquirido pela nota é apresentada a seguir.

Resolução 4.2: classe que representa a Nota de Entrada (Versão 1)

```
namespace ModelProject {
    class ProdutoNotaEntrada {
        public string CodigoProduto { get; set; }
```

```
        public string Descricao { get; set; }
        public double PrecoCusto { get; set; }
        public double QuantidadeComprada { get; set; }
    }
}
```

No processo relacionado ao registro de uma compra, as duas classes identificadas encaixam-se em um tipo de "relacionamento" chamado de *master-detail*, ou ainda, *pai-filhos*. Neste relacionamento, pode-se dizer que para cada objeto da classe `NotaEntrada` podem existir vários objetos da classe `ProdutoNotaEntrada`. Com esta informação, é preciso apenas definir como implementar essa **associação** entre essas classes. Neste primeiro momento, será feito o uso de uma matriz para isso. A seguir, apresento uma nova versão para a classe `NotaEntrada`.

Matrizes são comuns em situações práticas nas quais se necessita referenciar um grupo de variáveis do mesmo tipo, pelo mesmo nome simbólico. Podem ser vistas como um conjunto de variáveis do mesmo tipo, referenciáveis pelo mesmo nome e individualizadas entre si por meio de sua posição dentro desse conjunto. São conhecidas também como *variáveis indexadas*.

Resolução 4.3: classe NotaEntrada com a propriedade Produtos adicionada (Versão 2)

```
using System;
namespace ModelProject{
    class NotaEntrada {
        public string NumeroNota { get; set; }
        public string NomeFornecedor { get; set; }
        public string CNPJ { get; set; }
        public DateTime DataEmissao { get; set; }
        public DateTime DataEntrada { get; set; }
        public Produto[] Produtos { get; set; }
    }
}
```

A nova propriedade `Produtos` é uma matriz do tipo `Produto`. Na definição da propriedade, o uso dos colchetes (`[]`) caracteriza que ela será uma matriz. Em sua definição, não foi estipulado um tamanho físico para ela,

ou seja, quantos elementos (ou produtos) ela poderá receber como máximo. Esse tamanho será definido na sua inicialização.

É possível verificar com isso um limitador, que é a definição do tamanho para a propriedade que representa um conjunto/coleção de dados/objetos. Neste caso, ou se define um tamanho enorme, para evitar que em uma determinada situação não seja mais possível inserir novos elementos, ou é preciso saber quantos elementos serão inseridos nela, antes de utilizar. A segunda opção é a melhor, pois no caso da primeira, espaços reservados para possíveis elementos, que não forem usados, terão feito uso de recursos (espaço na memória) de maneira ineficiente.

Apesar do uso de matriz ser possível, as características apresentadas anteriormente trariam para o programador diversas situações que não fazem parte do problema, que é registrar a entrada de produtos. Buscando minimizar esses problemas e possibilitando ao programador dedicar todo o tempo de desenvolvimento (ou a maior parte dele) para a resolução específica do problema proposto, é recomendado o uso de coleções (`Collections`) quando se faz uso de associações.

4.3 Associações entre objetos

Anteriormente, na explicação sobre a "ligação" entre as classes `NotaEntrada` e `ProdutoNotaEntrada`, foi utilizado o termo *relacionamento*, uma expressão comum quando falamos em banco de dados. Entretanto, no paradigma orientado a objetos, o termo correto é **associação**, também comentado anteriormente.

Uma associação possui algumas características que são importantes para o momento da implementação. Para o exemplo trabalhado, a associação identificada é a de **composição**, que trabalha o conceito de **todo-parte**, no qual a parte tem seu tempo de vida diretamente ligado à parte todo. Isso significa que quando um objeto da classe todo (`NotaEntrada`) for removido, todos os objetos da classe parte (`ProdutoNotaEntrada`) também deverão ser, pois não faz sentido sua existência sem o objeto todo.

Em OO, uma **associação** representa um vínculo entre objetos de uma ou mais classes, de maneira que estes se relacionem. É possível que, por meio

desses vínculos (associações), um objeto invoque (ou requisite) serviços de outro objeto, ou que acesse ou atribua valores a propriedades deste. As associações podem ser unárias, ocorrendo em objetos da mesma classe; binárias, quando ocorrerem com dois objetos de classes distintas; e múltiplas, quando ocorrer em mais de dois objetos de classes distintas.

A **associação direta** ainda pode ser vista como o conceito de posse, em que, no exemplo apresentado, um objeto da classe `NotaEntrada` **tem um** conjunto de objetos da classe `ProdutoNotaEntrada`.

Composição é uma característica de uma associação, na qual um objeto é responsável por gerenciar o ciclo de vida de outros objetos. Fazendo uma análise mais refinada na associação entre `NotaEntrada` e `ProdutoNotaEntrada`, é possível constatar que um objeto da classe `NotaEntrada` **é composto** por objetos da classe `ProdutoNotaEntrada`. Não há motivos que justifiquem a existência de objetos da segunda classe sem que haja um objeto da primeira.

Para garantir as características de associação direta e composição nas resoluções apresentadas, é preciso detalhá-las um pouco mais. Para uma real aplicação da associação direta, duas novas classes serão criadas: `Fornecedor` e `Produto`. Também faz-se necessária a realização de alterações nas classes `NotaEntrada` e `ProdutoNotaEntrada`.

Apresento, a seguir, a classe `Fornecedor`.

Resolução 4.4: classe Fornecedor (Versão 1)

```
using System;

namespace ModelProject {
    class Fornecedor {
        public Guid Id { get; set; }
        public string Nome { get; set; }
        public string CNPJ { get; set; }
    }
}
```

Um `Guid` representa um número inteiro de 16 bytes, que pode ser usado sempre que for necessário um identificador que seja único, com pouca proba-

bilidade de existência de duplicação de seus valores. Normalmente, ele é utilizado para identificar computadores ou redes de computadores; entretanto, nada impede o seu uso na identificação de objetos.

A nova classe `Fornecedor` traz para si a responsabilidade de manter as propriedades referentes a fornecedores em vez de necessitar repetir esses dados em cada objeto `NotaEntrada`.

Na sequência, trago a classe `Produto`.

Resolução 4.5: classe Produto

```
namespace ModelProject {
    class Produto {
        public Guid Id { get; set; }
        public string Descricao { get; set; }
        public double PrecoDeCusto { get; set; }
        public double PrecoDeVenda { get; set; }
        public double Estoque { get; set; }
    }
}
```

Semelhante à classe `Fornecedor`, a classe `Produto` trouxe para si a responsabilidade das propriedades que dizem respeito a cada produto. Esses dados serão informados uma única vez por produto em vez de serem informados a cada registro de nota de entrada. E, no registro da nota de entrada, esses dados serão obtidos.

Apresento, na sequência, a nova implementação para a classe `ProdutoNotaEntrada`.

Resolução 4.6: classe ProdutoNotaEntrada com associação (Versão 2)

```
namespace ModelProject {
    class ProdutoNotaEntrada     {
        public Guid Id { get; set; }
        public Produto ProdutoNota { get; set; }
        public double PrecoCustoCompra { get; set; }
        public double QuantidadeComprada { get; set; }
```

```
    }
}
```

Na nova implementação da classe `ProdutoNotaEntrada`, é possível ver uma **associação direta** com a classe `Produto`, por meio da propriedade `ProdutoNota`. Verifica-se a existência de uma propriedade para guardar o preço de custo do produto na compra a ser registrada, enquanto o preço de custo da classe `Produto` tem o objetivo de registrar o custo atual para cada produto.

4.4 Composição entre classes fazendo uso de Collections

A estrutura `foreach` é responsável pela leitura de todos os elementos de uma coleção, associando cada um deles a uma variável, a cada iteração.

Finalizando o método, após o `foreach`, os valores obtidos nas variáveis "acumuladoras" são atribuídos à última linha do `grid`, devidamente formatados, assim como o valor para o `Label` ao lado do `Button`, que exibe o total consumido, sem os descontos.

4.5 Leitura de arquivo texto para cálculo de reajuste em folha de pagamento

A associação realizada entre as classes `NotaEntrada` e `ProdutoNotaEntrada` e apresentada na **Resolução 4.3** faz uso de matrizes e apenas foi exibida sem determinar como os objetos seriam registrados. Conforme também comentado, o mais indicado é o uso de `Collections`. Desta maneira, a **Resolução 4.7** traz a nova implementação para essa classe.

A biblioteca básica de classes do .NET, a *Base Class Library* (BCL), oferece um vasto conjunto de classes para coleções. Essas classes possibilitam um fácil gerenciamento de coleções de objetos, o que pode ser dificultoso quando se usa matrizes. Entretanto, é preciso conhecer cada uma delas para saber qual é a mais indicada para o problema a ser trabalhado. A escolha correta pode ser a chave para uma boa performance e manutenção de uma aplicação.

Resolução 4.7: classe NotaEntrada com associação por meio de uma coleção (Versão 3)

```
using System;
using System.Collections.Generic;
namespace ModelProject {
    class NotaEntrada {
        public Guid Id { get; set; }
        public string Numero { get; set; }
        public Fornecedor FornecedorNota { get; set; }
        public DateTime DataEmissao { get; set; }
        public DateTime DataEntrada { get; set; }
        public IList<ProdutoNotaEntrada> Produtos
            { get; set; }
    }
}
```

Verifica-se na declaração da propriedade `Produtos` que ela é definida como sendo `IList`, que é uma interface. Na declaração do tipo da coleção, ainda existe na sintaxe a definição do tipo de dado único que deve ser aceito nos elementos que vão compor a coleção `<Produto>`. Esta maneira para definição de coleções é conhecida como `Generics`.

Conceitualmente, uma coleção é semelhante a uma matriz. Porém, os benefícios trazidos por uma coleção são grandes. Esta opera sobre objetos da classe `Object`, ou seja, ela pode possuir qualquer tipo de objeto, ao contrário de uma matriz, que em sua declaração especifica o tipo de dado dos objetos que a vão compor.

Para tornar as coleções mais eficientes, surgiram os `Generics`, que definem que os objetos de uma coleção com essa característica sejam apenas do tipo especificado. Isso torna a coleção fortemente tipada, pois aceitará apenas esses objetos.

A associação por composição, quando implementada, precisa garantir sua característica, que é a de manter os objetos da "parte" apenas enquanto o "todo exista". Para o exemplo, os produtos da nota de entrada só deverão existir enquanto a nota existir, o que leva para a classe "todo" a responsabilidade pela manutenção desses dados. Para isso, uma nova implementação deve ser realizada, que está apresentada a seguir.

Resolução 4.8: classe NotaEntrada com métodos implementados (Versão 5)

```
using System;
using System.Collections.Generic;
namespace ModelProject {
    class NotaEntrada     {
        public Guid Id { get; set; }
        public string Numero { get; set; }
        public Fornecedor FornecedorNota { get; set; }
        public DateTime DataEmissao { get; set; }
        public DateTime DataEntrada { get; set; }
        public IList<ProdutoNotaEntrada> Produtos
            { get; set; }

        public NotaEntrada(){
            this.Produtos = new
                List<ProdutoNotaEntrada>();
        }

        public void RegistrarProduto(
                ProdutoNotaEntrada produto) {
            if (!this.Produtos.Contains(produto))
                this.Produtos.Add(produto);
        }

        public void RemoverProduto(
                ProdutoNotaEntrada produto) {
            this.Produtos.Remove(produto);
        }

        public void RemoverTodosProdutos() {
            this.Produtos.Clear();
        }
    }
}
```

Para que o método `RemoverProduto()` – que recebe uma instância do objeto `ProdutoNotaEntrada` – possa realmente remover o objeto

correto e para que, no método `RegistrarProduto()`, a busca realizada pela chamada `Contains()` possa também ser executada corretamente, é preciso definir na classe como ela deverá tratar a identidade de seus objetos. Esta implementação dá-se pela sobrescrita dos métodos `Equals()` e `GetHashCode()`, como mostro a seguir.

Resolução 4.9: classe ProdutoNotaEntrada com métodos Equals() e GetHashCode() (Versão 3)

```
namespace ModelProject {
    class ProdutoNotaEntrada {
        public Guid Id { get; set; }
        public Produto ProdutoNota { get; set; }
        public double PrecoCustoCompra { get; set; }
        public double QuantidadeComprada { get; set; }

        protected bool Equals(ProdutoNotaEntrada other) {
            return Id.Equals(other.Id);
        }

        public override bool Equals(object obj) {
            if (ReferenceEquals(null, obj))
                return false;
            if (ReferenceEquals(this, obj))
                return true;
            if (obj.GetType() !=
                    typeof (ProdutoNotaEntrada))
                return false;
            return Equals((ProdutoNotaEntrada) obj);
        }

        public override int GetHashCode() {
            return ProdutoNota.GetHashCode();
        }
    }
}
```

A implementação para o método `Equals()` apresentada tem um método a mais do que a que vimos no capítulo 3. Esta nova implemen-

tação é o método `Equals()` específico da classe, e não o sobrescrito (`override`). Nesse novo método, o objeto recebido é o de tipo desejado (`ProdutoNotaEntrada`), e não um `Object`. É possível verificar também que o método sobrescrito realiza uma chamada a esse novo método quando identifica que o tipo do objeto recebido é `ProdutoNotaEntrada`.

Pela forma como o método `Equals()` foi implementado, se o usuário inserir duas vezes o mesmo produto, essa operação será aceita, o que não é correto, pois deveria ser inserido uma vez o mesmo produto, com duas unidades na quantidade. Busque implementar o `Equals()` com base no produto.

As duas classes implementadas para a associação de composição devem ser vistas como serviços, ou seja, o modelo de negócio da aplicação. Para consumir esses serviços, é preciso pensar na camada em que o usuário interagirá com a aplicação, ou seja, a camada de visão, a interface com o usuário. A tela de uma nota de entrada ou uma página web são exemplos de implementação dessa camada.

Independente da tecnologia adotada para a camada de apresentação, a interação dela com a camada de modelo de negócio não deve ocorrer de maneira direta. Para isso, uma camada de controle é recomendada.

4.6 Criação dos projetos e definição de dependência entre eles

Para aplicar o conceito apresentado, crie uma nova solução. Nela, crie um novo projeto do tipo `Class Library` e dê a ele o nome de `ModelProject`. Nesse novo projeto, elimine o arquivo de classe criado e crie as classes: `Fornecedor`, `Produto`, `NotaEntrada` e `ProdutoNotaEntrada`, seguindo a implementação apresentada na última versão das respectivas resoluções. Esse projeto será responsável pela implementação da camada de modelo (ou de negócio).

Para implementar a camada de interação com o usuário (a de apresentação ou visão), crie na solução um novo projeto, agora do tipo `Windows Forms Application`, dando a ele o nome de `ViewProject`. Nesse novo projeto, remova o formulário criado automaticamente e crie três novos: `FormFornecedor`, `FormProduto` e `FormNotaEntrada`.

Para a persistência dos objetos informados nos formulários, será feito uso de coleções. Desta maneira, os dados existirão apenas enquanto a aplicação estiver em execução. Reforça-se que o importante neste momento é o conhecimento e a aplicação das técnicas e recursos para aplicação da associação de composição. O acesso ao banco de dados será apresentado e trabalhado, inicialmente, no capítulo 5. Sendo assim, crie um novo projeto `Class Library`, chamado `PersistenceProject`. Nele, crie uma classe chamada `Repository` e a implemente conforme apresentado na sequência.

Resolução 4.10: classe Repository

```
using System.Collections.Generic;
using ModelProject;

namespace PersistenceProject {
    public class Repository {
        private IList<Fornecedor> fornecedores =
            new List<Fornecedor>();
        private IList<Produto> produtos =
            new List<Produto>();
        private IList<NotaEntrada> notasEntrada =
            new List<NotaEntrada>();

        public Fornecedor InsertFornecedor(
                Fornecedor fornecedor) {
            this.fornecedores.Add(fornecedor);
            return fornecedor;
        }

        public void RemoveFornecedor(Fornecedor fornecedor) {
            this.fornecedores.Remove(fornecedor);
        }

        public IList<Fornecedor> GetAllFornecedores() {
            return this.fornecedores;
        }

        public Fornecedor UpdateFornecedor(
```

```
                Fornecedor fornecedor) {
            this.fornecedores[this.fornecedores.
                IndexOf(fornecedor)] = fornecedor;
            return fornecedor;
        }
    }
}
```

A classe Repository oferece métodos públicos para inserir e atualizar os objetos. O correto seria que essa classe e os controladores que fazem uso de seus serviços possuíssem, de maneira pública, apenas um método – neste exemplo, o SaveFornecedor() – e este invocasse o método de inserção ou atualização, que deverão ser privados, de acordo com o objeto que recebe. Isso porque a regra inicial para inserir um objeto é que ele não possua um ID.

Desta maneira, se o método SaveFornecedor() receber um objeto com ID, ele deve invocar o UpdateFornecedor(); caso contrário, o método chamado deve ser o InsertFornecedor(). Procure implementar essa técnica, mas veremos no capítulo 5 como se faz.

Essa resolução apresenta apenas métodos para os fornecedores. Crie, de maneira semelhante, os mesmos métodos para os outros campos (produtos e notasEntrada).

Aqui, é possível verificar que, nas importações de namespaces (usings), há a referência para o projeto de modelo, o ModelProject. A classe Repository está definida no namespace PersistenceProject.

Verifique que, por padrão, os namespaces das classes são idênticos aos nomes dos projetos, mas é claro que é possível mudar isso e definir namespaces específicos para cada uma. Em resumo, tem-se um projeto (PersistenceProject) fazendo uso de recursos (classes) de outro (ModelProject). O que ocorre de maneira mais técnica é que o primeiro referencia o segundo.

Para configurar a referência necessária, clique com o botão direito do mouse sobre o nome do projeto PersistenceProject e selecione a opção Add Reference. Na janela que é exibida, expanda a opção Solution e, em Projects, marque a caixa de seleção ao lado do nome do projeto ModelProject.

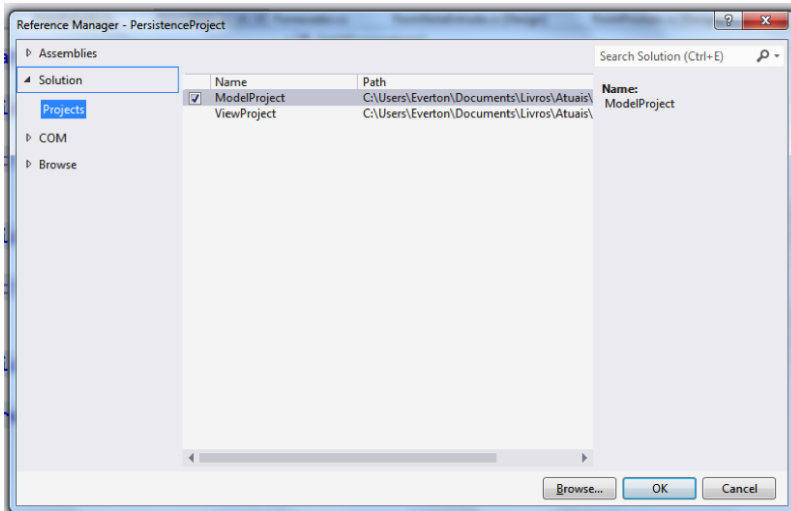



Fig. 4.1: Selecionando projeto para termos referência adicionada

Caso o projeto `ModelProject` não apareça na relação de projetos da solução, é preciso que seja realizado o build da solução.

Após confirmar a inclusão da referência clicando no botão `OK` na `Solution Explorer`, é possível verificar a existência do `ModelProject` no projeto `PersistenceProject`, em `References`.

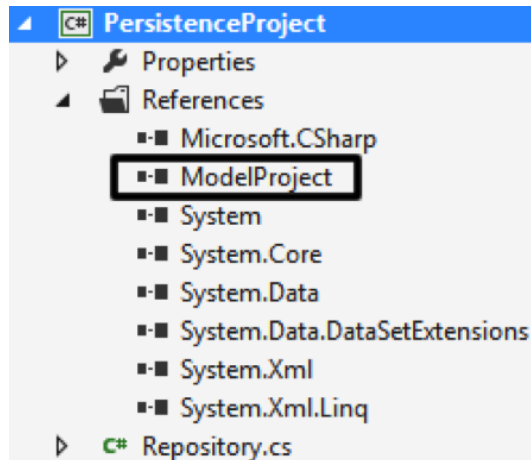



Fig. 4.2: Referência ao projeto ModelProject adicionada ao PersistenceProject

Nessa implementação de classe, mesmo adicionando a referência ao projeto `ModelProject` e inserindo a instrução `using ModelProject` no código, ao referenciar a classe `Fornecedor`, o compilador ainda acusará erro, não identificando essa classe. Isso ocorre porque a classe `Fornecedor` no `ModelProject` não tem nenhum modificador de escopo (ou de acesso). É preciso atualizar todas as classes que serão usadas em outros projetos para serem públicas (`public`). Apenas para exemplificar, a **Resolução 4.11** apresenta a mudança na classe `Fornecedor`.

Por meio de **modificadores de acesso** é possível definir como as classes e seus membros serão visíveis por outras classes e suas instâncias. Os modificadores disponíveis são: `public`, `private`, `protected` e `internal`. Quando uma classe é criada sem um modificador de escopo definido, é atribuído o `internal` por padrão. Por meio dele, as classes são visíveis apenas dentro do `assembly` que as define. No caso do exemplo, apenas o projeto que implementa a classe a reconhece. Uma classe definida em um projeto, e que será utilizada em outro, precisa ser definida como `public`.

Resolução 4.11: classe Fornecedor com modificador de acesso public (Versão 2)

```
namespace ModelProject {
    public class Fornecedor {
        public Guid Id { get; set; }
        public string Nome { get; set; }
        public string CNPJ { get; set; }
    }
}
```

Realize nas demais classes a alteração apresentada aqui, onde as classes recebam o modificador de escopo `public`.

4.7 Criando a camada de apresentação

No projeto criado anteriormente para a camada de apresentação, chamado `ViewProject`, no formulário `FormFornecedor`, feito para permitir a interação do usuário na manutenção dos dados relativos a fornecedores, implemente a janela apresentada na figura a seguir.

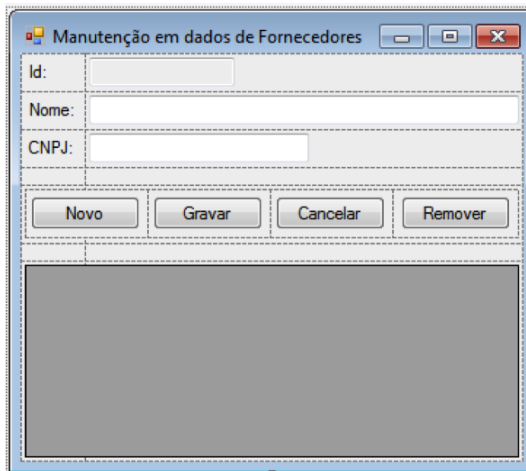



Fig. 4.3: Formulário para manutenção em dados de Fornecedores

O formulário desenhado e apresentado na figura 4.3 possui um componente `TableLayoutPanel`, dividido em linhas e colunas para uma melhor distribuição dos controles. Os `Labels` `Id`, `Nome` e `CNPJ` têm ao seu

lado `TextBoxs`, que permitirão a interação com o usuário, com exceção do `TextBox` do `Id`, que está definido como desabilitado (`Enabled = False`). Botões para as operações de manipulação nos dados também foram inseridos, como também um `DataGridView` ao final, que exibirá todos os fornecedores com dados já registrados.

No `SmartTag` do `DataGridView`, desmarque os `Checkboxs` conforme a figura 4.3. Atribua também à propriedade `SelectionMode` o valor `FullRowSelect`.

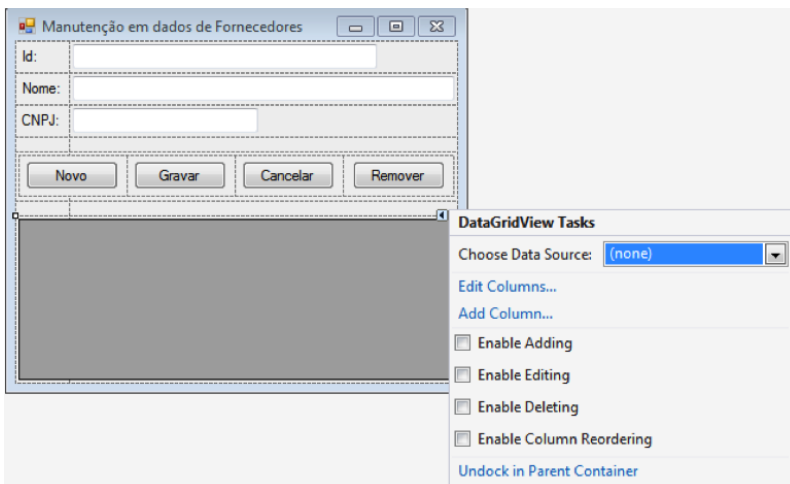



Fig. 4.4: Configuração de propriedades por meio do SmartTag do DataGridView

Os dados informados neste formulário serão inseridos na coleção `fornecedores`, definida na classe `Repository` no projeto `PersistenceProject`. Entretanto, a camada de apresentação não pode interagir de maneira direta com a camada de persistência, pois pode ocorrer de uma delas mudar; isso refletiria em muita manutenção.

Para evitar isso, será preciso criar um projeto que representará a camada controladora, fechando assim o MVC. Na solução, crie um novo projeto do tipo `Class Library` e o nomeie como `ControllerProject`. Elimine a classe criada por padrão com o projeto e crie uma classe chamada

`FornecedorController`, como apresentado na sequência.

Resolução 4.12: classe controladora para serviços relacionados à classe Fornecedor

```
using System.Collections;
using ModelProject;
using PersistenceProject;

namespace ControllerProject {
    public class FornecedorController {
        private Repository repository = new Repository();

        public Fornecedor Insert(Fornecedor fornecedor) {
            return this.repository.
                InsertFornecedor(fornecedor);
        }

        public void Remove(Fornecedor fornecedor)
            this.repository.RemoveFornecedor(fornecedor);
        }

        public IList<Fornecedor> GetAll() {
            return this.repository.GetAllFornecedores();
        }

        public Fornecedor Update(Fornecedor fornecedor) {
            return this.repository.UpdateFornecedor
                (fornecedor);
        }
    }
}
```

Aqui, apresento apenas o controlador para `Fornecedor`. Crie, de maneira semelhante, uma classe controladora para as classes `Produto` e `NotaEntrada`.

Note no código que existem as cláusulas `using ModelProject` e `using PersistenceProject`, o que indica a necessidade de incluir as referências para estes projetos no projeto `ControllerProject`.

A camada de apresentação precisa acessar a camada de controle, para que ela se responsabilize em delegar as responsabilidades para a camada de negócio (modelo) ou a de persistência. Desta maneira, atribua ao `ViewProject` a referência para o `ControllerProject`. E, na classe do formulário `FormFornecedor`, declare um campo para o controlador criado, como apresentado a seguir.

Resolução 4.13: declaração do controlador na camada de apresentação (Versão 1)

```
public partial class FormFornecedor : Form {
    private FornecedorController controller =
        new FornecedorController();
    // Código omitido
}
```

Com o controlador implementado, é preciso implementar os comportamentos dos botões disponíveis para a interação com o usuário. O primeiro botão a ser implementado será o `Gravar`. Veja, na sequência, a implementação para o método que captura o evento `Click` desse botão.

Resolução 4.14: método para o clique do botão Gravar (Versão 1)

```
private void btnGravar_Click(object sender, EventArgs e) {
    this.controller.Insert(
        new Fornecedor() {
            Id = Guid.NewGuid(),
            Nome = txtNome.Text,
            CNPJ = txtCNPJ.Text
        }
    );
}
```

A implementação para este método resume-se à invocação do método implementado no controlador para inserção de um fornecedor. Com exceção do `Id` para o fornecedor, que é gerado nesse método, os demais valores são obtidos diretamente dos controles da janela.

Neste momento, já é possível testar a aplicação. Realize um build para verificação de possíveis erros e, em caso de existência, corrija-os. Para executar a solução, clique com o botão direito sobre o nome do projeto `View Project` e, então, clique em `Set as Startup Project`. Na classe `Program`, defina o formulário `FormFornecedor.cs`, como o que será instanciado na execução da aplicação.

Nesse exemplo, a geração do valor para o `ID` está na invocação do método responsável pela inserção. Entretanto, o correto é que esse valor seja gerado no método que realizará a inserção efetiva, método este que faz parte da classe `Repository`. Busque implementar esta técnica.

Na janela que é exibida, como o controle que representa o `Id` está desabilitado, ele não recebe o foco, sendo possível informar apenas os valores de `Nome` e `CNPJ` para o fornecedor. Informe-os e clique no botão `Gravar`. Neste momento, um novo objeto foi inserido à coleção, entretanto, não há como visualizar esses dados. Para isso, é preciso atribuir a coleção ao `DataGridView`. Veja que apresento uma nova versão para o método desta resolução na sequência.

Resolução 4.15: método para o clique do botão Gravar com atualização do DataGridView (Versão 2)

```
private void btnGravar_Click(object sender, EventArgs e) {
    var fornecedor = this.controller.Insert(
        new Fornecedor() {
            Id = Guid.NewGuid(),
            Nome = txtNome.Text,
            CNPJ = txtCNPJ.Text
        }
    );
    txtID.Text = fornecedor.Id.ToString();
    dgvFornecedores.DataSource = null;
    dgvFornecedores.DataSource = this.controller.GetAll();
}
```

Ao contrário do apresentado em exemplos anteriores, nos quais a coleção atribuída ao `DataSource` do `DataGridView` era um `BindingList`, neste

exemplo prevalece o retorno do método `GetAll()`, que é um `IList`. Isso torna tanto o controlador quanto o repositório mais independentes.

Caso seja de interesse (ou necessidade da camada de apresentação), a transformação desse retorno para um `BindingList` é simples, basta apenas instanciar a classe passando o retorno de `GetAll()` como parâmetro para o construtor. Pelo fato de a coleção atribuída ao `DataSource` não causar a atualização imediata dos controles em que ela está ligada, para que cada fornecedor inserido na coleção possa aparecer no `DataGridView`, torna-se necessário atribuir `null` ao `DataSource` e, em seguida, à nova coleção de dados, como se fosse um *reset* de dados no controle.

Neste exemplo, a geração do valor para o `ID` está na invocação do método responsável pela inserção. Entretanto, o correto é que esse valor seja gerado no método que realizará a inserção efetiva, método este que faz parte da classe `Repository`. Busque implementar esta técnica.

Após a inserção dos dados na coleção e a exibição atualizada dela no `DataGridView`, os dados informados também continuam aparecendo nos controles em que foram informados. Neste momento, caso o usuário deseje informar um novo fornecedor, é importante que os controles estejam sem dados. A seguir, apresento a implementação do método que captura o evento `Click` do botão `Novo`.

Resolução 4.16: método para o clique do botão Novo (Versão 1)

```
private void BtnNovo_Click(object sender, EventArgs e) {
    txtID.Text = string.Empty;
    txtNome.Text = string.Empty;
    txtCNPJ.Text = string.Empty;
}
```

O próximo passo é permitir que o usuário altere dados já informados e armazenados na coleção. A maneira adotada é que o registro (linha) selecionado no `DataGridView` terá os dados atualizados nos `TextBoxs`, permitindo, assim, sua alteração. Como o `DataGridView` está configurado para selecionar a linha inteira por meio do clique do mouse, é importante identificar qual evento deve ser capturado. Para este caso, o evento escolhido é

o SelectionChanged. Na sequência, há a implementação do método que captura o evento identificado.

Resolução 4.17: método que captura o evento SelectionChanged do DataGridView

```
private void dgvFornecedores_SelectionChanged(
        object sender, EventArgs e) {
    txtID.Text = dgvFornecedores.CurrentRow.Cells[0].
        Value.ToString();
    txtNome.Text = dgvFornecedores.CurrentRow.Cells[1].
        Value.ToString();
    txtCNPJ.Text = dgvFornecedores.CurrentRow.Cells[2].
        Value.ToString()
}
```

Com a implementação deste método, os controles TextBoxs serão sempre atualizados quando ocorrer a gravação dos dados também, e não apenas quando o usuário selecionar uma nova linha do DataGridView. Sendo assim, é importante que, ao inserir um novo fornecedor na coleção, os controles sejam "limpos" e que o DataGridView não tenha nenhuma linha selecionada. Para isso, adapte o método que captura o clique do botão Gravar. Mostro essa implementação a seguir.

Resolução 4.18: método que captura o evento Click do botão Gravar (Versão 3)

```
private void btnGravar_Click(object sender, EventArgs e) {
    var fornecedor = this.controller.Insert(
        new Fornecedor() {
            Id = Guid.NewGuid(),
            Nome = txtNome.Text,
            CNPJ = txtCNPJ.Text
        }
    );
    dgvFornecedores.DataSource = null;
    dgvFornecedores.DataSource = this.controller.GetAll();
    ClearControls();
}
```

Verifica-se na nova versão para este método que a instrução responsável pela atribuição do `Id` ao respectivo `TextBox` foi retirada, e que foi inserida uma chamada a um novo método, nomeado `ClearControls()`, apresentado a seguir.

Resolução 4.19: método para limpeza de controles do formulário

```
private void ClearControls() {
    dgvFornecedores.ClearSelection();
    txtID.Text = string.Empty;
    txtNome.Text = string.Empty;
    txtCNPJ.Text = string.Empty;
    txtNome.Focus();
}
```

A primeira linha deste novo método faz com que não exista nenhuma linha selecionada no `DataGridView`. Com isso, caso o usuário clique em qualquer linha, o evento disparado permitirá a atualização correta dos `TextBoxs`, uma vez que, sempre que há a ligação do controle com uma fonte de dados por meio do `DataSource`, o primeiro registro é marcado como selecionado no `DataGridView`. As três instruções seguintes são as mesmas implementadas no método que captura o evento `Click` do botão `Novo`. O método termina atribuindo o foco da aplicação ao primeiro controle editável do formulário, o nome do fornecedor.

O método `ClearSelection()` retira qualquer seleção existente em linhas do `DataGridView`. Esta operação é interessante, uma vez que, quando uma linha for selecionada, os dados dela (registro/objeto) são exibidos nos controles de interação.

É possível trazer para o método `ClearControls()` as instruções de atualização do `DataGridView` que fazem parte do método `Click` do botão `Gravar`. Caso tente depurar a aplicação, acompanhando o método que captura o evento `SelectionChanged`, será possível verificar que, a cada alteração no controle, esse método é invocado. Isso porque a cada mudança, o evento é disparado.

Com esta nova implementação, o método que captura o evento `Click` do botão `Novo` também precisa ser alterado. Este pode ser visualizado na

sequência. Note que, desta maneira, está sendo propiciada a reutilização de código, criando um método específico para uma atividade requerida em diversas "partes" da aplicação.

Resolução 4.20: método que captura o evento Click do botão Novo (Versão 4)

```
private void BtnNovo_Click(object sender, EventArgs e) {
    ClearControls();
}
```

Com a possibilidade de alteração dos dados de um fornecedor já cadastrado, surge um novo problema. Insira um fornecedor, selecione-o no `DataGridView` e clique em gravar. Note que um novo fornecedor foi inserido com um novo `Id`, quando o que era desejado era apenas a atualização dos dados, com a manutenção do `Id` original. Para isso, mudanças devem ser realizadas no método que captura o `Click` do botão `Gravar`. Veja a nova implementação para este método a seguir.

Resolução 4.21: método que captura o evento Click do botão Gravar (Versão 4)

```
private void btnGravar_Click(object sender, EventArgs e) {
    var fornecedor = new Fornecedor() {
        Id = (txtID.Text == string.Empty ?
            Guid.NewGuid() : new Guid(txtID.Text)),
        Nome = txtNome.Text,
        CNPJ = txtCNPJ.Text
    };
    fornecedor = (txtID.Text == string.Empty ?
        this.controller.Insert(fornecedor) :
        this.controller.Update(fornecedor));
    dgvFornecedores.DataSource = null;
    dgvFornecedores.DataSource = this.controller.GetAll();
    ClearControls();
}
```

O operador ternário tem a funcionalidade de validar uma simples instrução `if...else`, onde, dada uma condição, retorna-se um valor para `true`

ou false. É chamado de **ternário** por possuir três operandos separados pelos operadores ? e :. Sua sintaxe é <condição a ser avaliada> ? <valor_para_verdadeiro> : <valor_para_falso>.

Observe que o fornecedor agora é criado fora da chamada ao método Insert() e que, a propriedade Id pode ter seu valor atribuído de duas maneiras: na primeira, gerando um novo, caso o TextBox do Id esteja vazio; a segunda é obtendo o Id existente no TextBox, que existirá apenas se um fornecedor for selecionado no DataGridView. Para essa atribuição, foi utilizado o operador condicional ternário, o ?:. Esse mesmo operador é usado na invocação do método que gerará a operação de inserção ou atualização.

Para que a atualização do fornecedor enviado como argumento possa efetivamente ocorrer, é preciso sobrescrever os métodos Equals() e GetHashCode(), pois é por meio deles que se identifica um objeto, utilizada pelo método IndexOf() (e diversos outros que realizam buscas na coleção), na determinação de qual objeto deve ser buscado na coleção. A seguir, apresento a nova implementação para a classe Fornecedor.

Resolução 4.22: classe Fornecedor com métodos para identidade de objetos (Versão 3)

```
using System;

namespace ModelProject {
    public class Fornecedor {
        public Guid Id { get; set; }
        public string Nome { get; set; }
        public string CNPJ { get; set; }

        protected bool Equals(Fornecedor other) {
            return Id.Equals(other.Id);
        }

        public override bool Equals(object obj) {
            if (ReferenceEquals(null, obj)) return false;
            if (ReferenceEquals(this, obj)) return true;
            if (obj.GetType() != typeof (Fornecedor))
                return false;
```

```
            return Equals((Fornecedor) obj);
        }

        public override int GetHashCode() {
            return
        }
}
```

Na sequência, o botão `Remover` possuirá a responsabilidade de retirar da coleção de fornecedores aquele que estiver atualmente sendo exibido no formulário. Para isso, será preciso garantir que exista algum fornecedor com dados exibidos no formulário, antes de permitir sua remoção. Na sequência, apresento o método que captura o evento `Click` do botão `Remover`.

Resolução 4.23: método que captura o evento Click do botão Remover

```
private void BtnRemover_Click(object sender, EventArgs e){
    if (txtID.Text == string.Empty) {
        MessageBox.Show(
            "Selecione o FORNECEDOR a ser removido no GRID");
    }
    else {
        this.controller.Remove(
            new Fornecedor() {
                Id = new Guid(txtID.Text)
            }
        );
        dgvFornecedores.DataSource = null;
        dgvFornecedores.DataSource =
            this.controller.GetAll();
        ClearControls();
    }
}
```

Note que as três últimas linhas do método são idênticas às do que captura o evento `Click` do botão `Gravar` (visto na **Resolução 4.18**). A exceção está no objeto fornecedor enviado ao método `Remove()`, que possui apenas o

Id. Uma vez que a identidade dos objetos está nesta propriedade, as demais não são necessárias.

Outra abordagem seria remover um fornecedor com base apenas no Id, não enviando um objeto, mas sim apenas o valor. Fica o desafio para a implementação de um método que possa encapsular as instruções comuns entre o método para gravar e remover ou, avaliar se essas instruções podem fazer parte do método ClearControls().

Concluindo as funcionalidades para esta janela, a seguir represento o método que captura o evento Click do botão Cancelar.

Resolução 4.24: método que captura o evento Click do botão Cancelar

```
private void BtnCancelar_Click(object sender,
        EventArgs e) {
    ClearControls();
}
```

4.8 Criando uma janela principal para a aplicação e menus para acesso aos formulários criados

Com a criação do formulário para Produtos, indicada no final do item anterior, o projeto ViewProject possuirá dois formulários. Após a implementação dos formulários para as demais classes já identificadas, este número aumentará.

Não é conveniente nem para testes ficar alterando o formulário de inicialização na classe Program a cada execução. Por isso, vamos criar um formulário, representando uma janela principal para toda a aplicação. Nela, haverá um menu com acesso a todas as janelas existentes na aplicação. No projeto ViewProject, crie um formulário chamado JanelaPrincipal.

Neste novo formulário, algumas propriedades devem ser configuradas, conforme mostrado a seguir.

- **Propriedade:** Name

- **Valor para atribuir:** FormJanelaPrincipal

- **Propriedade:** WindowState
- **Valor para atribuir:** Maximized

Esta propriedade define o estado da janela no momento em que for exibida pela primeira vez. Os valores são `Maximized`, o que faz com que a janela seja exibida maximizada; `Minimized`, que a torna minimizada; e `Normal` (padrão), que faz com que a janela seja exibida da maneira que ela é desenhada.

- **Propriedade:** Text
- **Valor para atribuir:** Janela principal da aplicação

- **Propriedade:** MaximizeBox
- **Valor para atribuir:** False

Por meio desta propriedade, define-se se o botão que maximiza uma janela (ou restaura seu tamanho inicial) é exibido ou não. Normalmente, quando se exibe a janela maximizada, não se deseja que o usuário mude seu tamanho. Desta maneira, é comum quando a propriedade `WindowState` esteja em `Maximized` e que a `MaximizeBox` seja `False`.

Com a janela criada e configurada, é preciso inserir nela o controle responsável pelo menu de acesso aos formulários. Esse controle, chamado `MenuStrip`, pode ser encontrado na categoria `Menus & Toolbars`, como pode ser verificado por meio da figura a seguir.

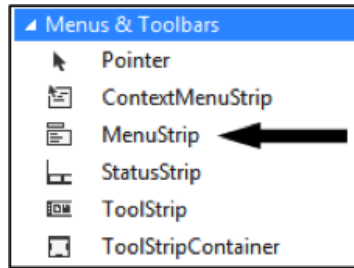



Fig. 4.5: Categoria de controles para menus e barras de tarefas

Arraste o controle `MenuStrip` para a área de desenho do formulário, e seu formulário deverá ficar semelhante ao apresentado:

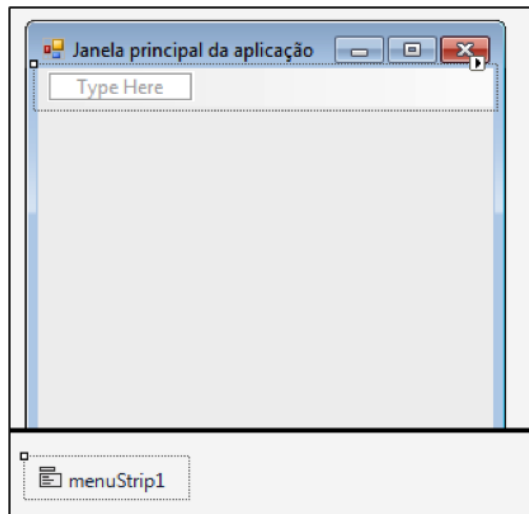



Fig. 4.6: Controle MenuStrip arrastado para o formulário

O controle `MenuStrip` adiciona uma barra de menus a um formulário de uma aplicação `Windows Forms`. Por meio desse controle, é possível adicionar uma área de menus e, então, adicionar menus ou criar menus personalizados, diretamente por meio do Visual Studio.

A princípio, não há necessidade de configuração em nenhuma proprie-

dade desse controle. Entretanto, é preciso inserir as opções que o menu principal da janela oferecerá. Para isso, na parte com o texto `Type Here`, digite **Cadastros**. Ao realizar essa digitação, um novo espaço também com o texto `Type Here` será exibido ao lado direito do menu `Cadastros`, digite **Registros**. Seu formulário deve estar semelhante ao apresentado:

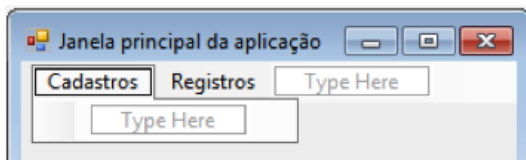



Fig. 4.7: Opções para o menu principal

As opções de menu `Cadastros` e `Registros` terão novas opções ligadas a elas, pois são vistas como categorias. Para inserir uma opção, abaixo de `Cadastros`, onde está escrito `Type Here`, basta digitar a desejada. A figura 4.8 apresenta essas opções:

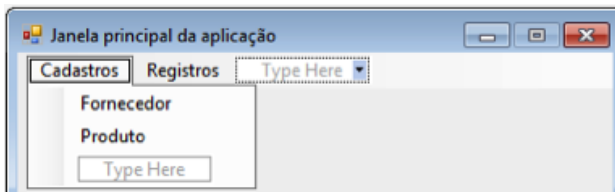



Fig. 4.8: Criação de opções para o menu Cadastros

Com os menus e suas opções criados, faz-se necessária a implementação para o acesso ao formulário desejado. Assim como os botões, os menus possuem um evento `Click`, e é nele que deve ser implementado o código que exibirá a janela desejada ao usuário. Para criar e acessar o método responsável pelo menu `Click` de um menu – que é um controle `ToolStripMenuItem` –, pode-se realizar um duplo clique no item desejado, ou, por meio da `Properties Window` em eventos, acessar o evento `Click`. Na sequência, represento o método que captura o evento `Click` para o menu

Fornecedores.

Resolução 4.25: método que captura o evento Click do menu Fornecedores

```
private void fornecedorToolStripMenuItem_Click(
        object sender, EventArgs e) {
    new FormFornecedor().ShowDialog();
}
```

Realize o build na solução e execute-a. Na janela principal, selecione o menu de cadastros e, em seguida, o do fornecedor. A janela que representa o formulário de fornecedor será exibida. Faça o mesmo para a opção de menu referente a produtos.

O controle ToolStripMenuItem representa uma opção selecionável, que pode ser exibida por um MenuStrip ou ContextMenuStrip. Pode ser visto como itens de um menu.

Com a implementação da classe que representa a janela principal da aplicação, os formulários agora não são executados diretamente, mas sim invocados por outro formulário. Desta maneira, é possível pensar em uma injeção de dependência. Cada formulário criado possui seu controlador, que é declarado e instanciado na própria classe do formulário. É possível que exista mais de um formulário que precise do mesmo controlador, o que redundaria código, possibilitando inclusive perda de performance e concorrência dos dados. Uma vez que a janela principal será a responsável por invocar todos os formulários, os controladores poderiam ser declarados nela e ela fornecer ao consumidor (os formulários) os controladores já "prontos" para o uso. A seguir, apresento a declaração dos controladores na classe JanelaPrincipal.

Resolução 4.26: declaração de controladores no formulário da Janela Principal

```
public partial class FormJanelaPrincipal : Form {
    private FornecedorController fornecedorController =
        new FornecedorController();
    private ProdutoController produtoController =
        new ProdutoController();
```

```
    // Código omitido
}
```

Com os controladores definidos, é preciso injetá-los nos formulários que os utilizarão. Na sequência, apresento o novo código para o método que captura o evento `Click` do menu de `Fornecedores`. É preciso alterar também o menu para `Produtos`.

Resolução 4.27: nova versão para o método que captura o evento Click do menu de Fornecedores

```
private void fornecedorToolStripMenuItem_Click(
        object sender, EventArgs e) {
    new FormFornecedor(fornecedorController).
        ShowDialog();
}
```

Observe na chamada ao construtor da classe `FormFornecedor` o envio de um argumento, que é o controlador. Desta maneira, é preciso também alterar o código da classe `FormFornecedor`. A seguir, mostro a nova implementação.

Resolução 4.28: nova versão para a classe FormFornecedor (Versão 2)

```
public partial class FormFornecedor : Form {
    private FornecedorController controller;
    public FormFornecedor(
            FornecedorController controller) {
        InitializeComponent();
        this.controller = controller;
    }
    // Código omitido
}
```

Injeção de dependência (ID – *Dependency Injection*) é um termo também conhecido como **Inversão de Controle** (IoC – *Invertion of Control*). É

um padrão de projeto que busca remover o forte acoplamento entre componentes que sejam dependentes – como o controlador e a camada de apresentação. A ID permite facilitar a concepção e implementação, fazendo uso de baixo acoplamento de objetos reutilizáveis e testáveis. A IoC também fortalece a recomendação para que o controle de instância de uma determinada classe – como no caso do `controller` – seja tratado externamente, como na classe `JanelaPrincipal`, e não dentro da classe consumidora, como a `FormFornecedor`. Desta maneira, ocorre a inversão de controle, por meio da injeção de dependência.

4.9 Associando objetos na implementação da Nota de Entrada

Para a aplicação de associação entre objetos, será feito uso da classe `NotaEntrada`, associada a `Fornecedor`, e da classe `ProdutoNotaEntrada`, associada a `NotaEntrada` e `Produto`. A leitura dessas associações pode ser: *uma nota de entrada é emitida por um fornecedor, ela possui um ou vários produtos comprados e cada produto comprado refere-se a um produto existente.*

A implementação da interface com o usuário é simples, pois o foco é a associação. Ela está representada pela figura a seguir:

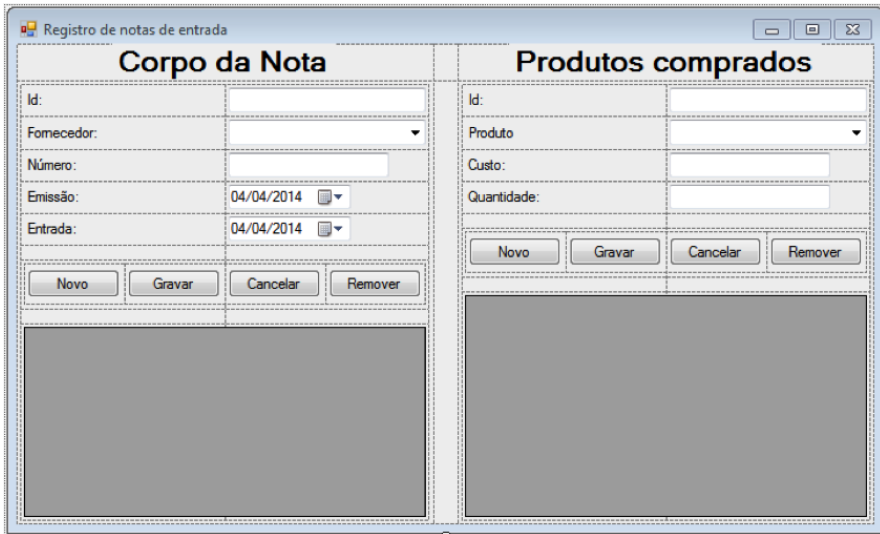



Fig. 4.9: Formulário que representa a Nota de Entrada de Produtos

Por meio da figura 4.9, verifica-se a existência de duas regiões: uma para o corpo da nota, representado pela classe `NotaEntrada`; e outra para os produtos que compõem cada nota, representado pela classe `ProdutoNotaEntrada`.

Na parte que compete ao corpo da nota, verifica-se a existência de um dado referente ao `Fornecedor`, e na dos produtos comprados, verifica-se também a existência de um dado referente ao `Produto`. Abaixo desses campos que terão os dados informados, existem dois grids: um para as notas registradas e um para os produtos registrados para cada nota, aparecendo sempre os que pertencem à nota selecionada.

Os dados referentes ao `Fornecedor` e aos `Produtos` serão registrados por meio de controles `ComboBox`, que estão na categoria `Common Controls` e, inicialmente, não precisam de nenhuma configuração. Esses controles armazenam uma coleção de dados – objetos, neste caso – e permitem o usuário selecionar um deles. Essa coleção de dados, no caso `Produtos` e `Fornecedores`, deverá estar disponível no momento em que o usuário visualizar a janela de notas de entrada. Desta maneira, o formulário de nota de

entrada deverá receber os controladores de produtos e fornecedores, para então fornecer os respectivos dados aos `ComboBoxs`.

A princípio, configure nos `ComboBoxs` apenas a propriedade `DropDownStyle` para `DropDownList`, para evitar que o usuário digite valores na lista. Os `DateTimePickers`, para as datas de emissão e entrada, têm apenas a propriedade `Format` configurada para `Short`. Os demais componentes são todos já conhecidos e utilizados em exemplos anteriores. Na sequência, apresento o código inicial da classe que representa o `FormNotaEntrada`.

Resolução 4.29: declaração de controladores para a classe FormNotaEntrada

```
public partial class FormNotaEntrada : Form {
    private NotaEntradaController controller;
    private FornecedorController fornecedorController;
    private ProdutoController produtoController;

    private NotaEntrada notaAtual;

    public FormNotaEntrada(
        NotaEntradaController controller,
                FornecedorController
                fornecedorController,
                ProdutoController produtoController) {
        InitializeComponent();
        this.controller = controller;
        this.fornecedorController =
            fornecedorController;
        this.produtoController = produtoController;
        InicializaComboBoxs();
    }
    // Código omitido
}
```

O `ComboBox` é um controle que possui uma lista suspensa com itens que podem ser selecionados. Essa lista pode ser exibida ou ocultada por meio de uma interação com o usuário, normalmente com o clique do mouse sobre a

seta existente no controle. Pelo teclado, o controle precisa estar com o foco e, então, pressiona-se a combinação da tecla de espaço e seta para baixo. Sua aparência pode ser configurada por meio da propriedade `DropDownStyle`, e os itens da lista suspensa são armazenados e recuperados na propriedade `Items`.

Com exceção à última linha do construtor, todo o código é facilmente interpretado. Inicialmente, declaram-se os controladores, depois os campos a serem utilizados e, em seguida, o construtor que registra os controladores e chama o método (última linha do construtor) responsável por inicializar os `ComboBoxs` de `Fornecedores` e `Produtos`, o `InicializaComboBoxs()`. Este é exibido na sequência.

Resolução 4.30: método responsável pela inicialização dos ComboBoxs

```
private void InicializaComboBoxs() {
    cbxFornecedor.Items.Clear();
    cbxProduto.Items.Clear();

    foreach (Fornecedor fornecedor in
            this.fornecedorController.GetAll()) {
        cbxFornecedor.Items.Add(fornecedor);
    }

    foreach (Produto produto in
            this.produtoController.GetAll()) {
        cbxProduto.Items.Add(produto);
    }
}
```

O método `InicializaComboBoxs()`, inicialmente, *limpa* o conteúdo dos `ComboBoxs` e depois os "popula", por meio dos métodos `GetAll()` de seus controladores. Note que a propriedade `Items` é uma coleção com um método `Add()`, que recebe os objetos `Fornecedores` e `Produtos`, recuperados. Para testar, é preciso inserir uma opção no menu `Registros` para acessar a nota de entrada. Crie essa opção, como exibido por meio da figura a seguir:

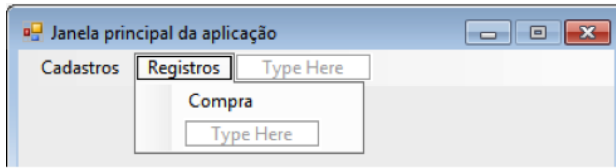



Fig. 4.10: Opção de menu para Compra registrada no menu Registros

Na sequência, apresento o código para que a janela de compra seja invocada.

Resolução 4.31: método que captura o evento Click da opção Compra do Menu Registros

```
private void compraToolStripMenuItem_Click(object sender,
        EventArgs e) {
    new FormNotaEntrada(notaEntradaController,
        fornecedorController, produtoController).
        ShowDialog();
}
```

Para testar a implementação realizada até este momento, execute sua aplicação, acesse os formulários de fornecedores e produtos, registre alguns itens e, então, acesse o formulário de compras e clique no `ComboBox` de `Fornecedor` ou `Produto`. A figura 4.11 apresenta a janela com o `ComboBox` de `Fornecedor` exibindo os fornecedores registrados.

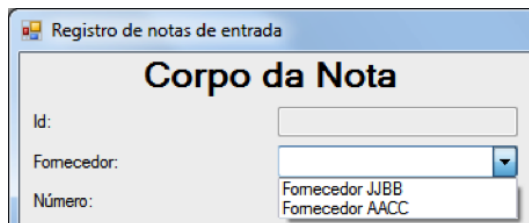



Fig. 4.11: ComboBox exibindo fornecedores previamente registrados

4.10 Implementando a interação do usuário com o registro do Corpo da Nota de Compra

Com o formulário todo "desenhado", é preciso implementar os comportamentos para cada botão. Inicialmente, os botões que serão implementados referem-se ao corpo da nota, ou seja, os dados referentes à classe `NotaEntrada`. A seguir, apresento o código implementado para o método que implementa o botão `Novo`, da nota de entrada.

Resolução 4.32: método que captura o evento Click do botão Novo do corpo da Nota

```
private void BtnNovoNota_Click(object sender,
        EventArgs e) {
    ClearControlsNota();
}
```

O método `ClearControlsNota()` tem o mesmo objetivo do método `ClearControls()`, apresentado na criação do formulário de fornecedores. Mostro, a seguir, sua implementação.

Resolução 4.33: método que realiza a preparação dos controles do corpo da nota para uma nova inserção de dados

```
private void ClearControlsNota() {
    dgvNotasEntrada.ClearSelection();
    dgvProdutos.ClearSelection();
    txtIDNotaEntrada.Text = string.Empty;
    cbxFornecedor.SelectedIndex = -1;
    txtNumero.Text = string.Empty;
    dtpEmissao.Value = DateTime.Now;
    dtpEntrada.Value = DateTime.Now;
    cbxFornecedor.Focus();
}
```

Nesse código, existe um ponto que ainda não foi apresentado, que é a atribuição de valor à propriedade `SelectedIndex` do `ComboBox` de for-

necedores. Atribuindo o valor -1 à ela, assume-se que não é para apresentar nenhum item como selecionado, o que é interessante em um processo de inserção, no qual o usuário obrigatoriamente precisará escolher um item, no caso, um fornecedor.

A segunda implementação a ser apresentada para o corpo da nota de entrada refere-se o botão `Gravar`, que tem seu código mostrado na sequência.

Resolução 4.34: método que captura o evento Click do botão Gravar do corpo da nota de entrada

```
private void btnGravar_Click(object sender, System.
        EventArgs e) {
    var notaEntrada =  new NotaEntrada() {
        Id = (txtIDNotaEntrada.Text ==
            string.Empty ? Guid.NewGuid() :
                new Guid(txtIDNotaEntrada.Text)),
        DataEmissao = dtpEmissao.Value,
        DataEntrada = dtpEntrada.Value,
        FornecedorNota = (Fornecedor) cbxFornecedor.
            SelectedItem,
        Numero = txtNumero.Text
    };
    notaEntrada = (txtIDNotaEntrada.Text ==
        string.Empty ? this.controller.Insert(
        notaEntrada) : this.controller.Update(
        notaEntrada));
    dgvNotasEntrada.DataSource = null;
    dgvNotasEntrada.DataSource = this.controller.
        GetAll();
    ClearControlsNota();
}
```

A primeira instrução do comportamento para o método apresentado faz referência ao método `Insert()` – que será apresentado na **Resolução 4.35** – do controlador específico para a classe `NotaEntrada` e seu respectivo formulário. Também para o caso de a nota estar sendo alterada e não inserida, o método `Update()` da **Resolução 4.36** é invocado, quando necessário.

Outra nova instrução é a chamada à propriedade `SelectedItem` do `ComboBox`, que retorna o objeto atualmente selecionado, neste caso, o fornecedor selecionado para a nota de compra a ser registrada. Ao final, o método `GetAll()` – da **Resolução 4.37** – também do controlador para a nota de entrada é invocado. Para a criação da classe controladora para nota de entrada, siga as orientações fornecidas para a implementação da classe `FornecedorController`.

A propriedade `SelectedItem` do `ComboBox` permite atribuir e recuperar o item (objeto) atualmente selecionado. Caso o objeto seja atribuído e ele exista na lista de itens disponíveis, há uma atualização no valor da propriedade `SelectedIndex`, que recebe o valor do índice onde o objeto atribuído se encontra dentro dos itens disponíveis.

Resolução 4.35: método Insert() da classe NotaEntradaController

```
public NotaEntrada Insert(NotaEntrada notaEntrada) {
    return this.repository.InsertNotaEntrada
        (notaEntrada);
}
```

Resolução 4.36: método Update() da classe NotaEntradaController

```
public NotaEntrada Update(NotaEntrada notaEntrada) {
    return this.repository.UpdateNotaEntrada(
        notaEntrada);
}
```

Resolução 4.37: método GetAll() da classe NotaEntradaController

```
public IList<NotaEntrada> GetAll() {
    return this.repository.GetAllNotasEntrada();
}
```

Como pode ser verificado nas **Resoluções 4.35**, **4.36** e aqui, elas fazem referências a métodos de um repositório, que é declarado de maneira semelhante ao realizado na classe `FornecedorController`. As

Resoluções 4.38, **4.39** e **4.40** exibem a implementação, respectivamente, dos métodos `InsertNotaEntrada()`, `UpdateNotaEntrada()` e `GetAllNotasEntrada()`, declarados na classe `Repository`.

Resolução 4.38: método InsertNotaEntrada() da classe Repository

```
public NotaEntrada InsertNotaEntrada(
        NotaEntrada notaEntrada) {
    this.notasEntrada.Add(notaEntrada);
    return notaEntrada;
}
```

Resolução 4.39: método UpdateNotaEntrada() da classe Repository

```
public NotaEntrada UpdateNotaEntrada(
        NotaEntrada notaEntrada) {
    this.notasEntrada[this.notasEntrada.
        IndexOf(notaEntrada)] = notaEntrada;
    return notaEntrada;
}
```

Resolução 4.40: método GetAllNotasEntrada() da classe Repository

```
public IList<NotaEntrada> GetAllNotasEntrada() {
    return this.notasEntrada;
}
```

O método `InsertNotaEntrada()` – tanto no repositório quanto no controlador – retornam um objeto `NotaEntrada` e, ao verificar o código, não se vê o motivo para isso. Entretanto, esta técnica é importante, pois na camada de persistência (`Repository`) deve ocorrer a atribuição do `ID` para cada objeto persistido (armazenado), o que não ocorre na aplicação apresentada. Porém, prevendo essa necessidade, os métodos já têm esse retorno em suas assinaturas.

A terceira implementação para o corpo da nota de entrada refere-se ao botão `Cancelar`, que tem seu código apresentado na sequência.

Resolução 4.41: método que captura o evento Click do botão Cancelar do corpo da nota

```
private void BtnCancelarNota_Click(object sender,
        EventArgs e) {
    ClearControlsNota();
}
```

Aqui, verifica-se que o comportamento para esse botão é o mesmo que o implementado para o botão `Novo`. A quarta implementação, mostrada a seguir, refere-se ao comportamento para o clique no botão `Remover` da nota.

Resolução 4.42: método que captura o evento Click do botão Remover do corpo da nota

```
private void BtnRemoverNota_Click(object sender,
        EventArgs e) {
    if (txtIDNotaEntrada.Text == string.Empty) {
        MessageBox.Show(
            "Selecione a NOTA a ser removida no GRID");
    }
    else {
        this.controller.Remove(
            new NotaEntrada() {
                Id = new Guid(txtIDNotaEntrada.Text)
            }
        );
        dgvNotasEntrada.DataSource = null;
        dgvNotasEntrada.DataSource =
        this.controller.GetAll();
        ClearControlsNota();
    }
}
```

Para que uma nota seja removida, é preciso que seus dados estejam exibidos para que valide a sua existência. Esta verificação é realizada por meio da existência de um valor para o `ID` da nota, que é gerado quando ela é gravada pela primeira vez. Na implementação do método, é possível identificar essa

avaliação que, caso seja confirmada, invoca o método `Remove()` do controlador. O objeto enviado para o método, que representa uma nota de entrada, recebeu apenas o `ID`, pois a identidade do objeto (métodos `Equals()` e `GetHashCode()`) baseia-se nesse campo.

A implementação do método `Remove()` pode ser verificada na sequência.

Resolução 4.43: método Remove() da classe NotaEntradaController

```
public void Remove(NotaEntrada notaEntrada) {
    notaEntrada.RemoverTodosProdutos();
    this.repository.RemoveNotaEntrada(notaEntrada);
}
```

Semelhante aos demais métodos implementados na classe controladora, o método `Remove()`, que recebe a nota de entrada, invoca o método `RemoveNotaEntrada()`, implementado na classe `Repository` e que é apresentando a seguir. Porém, antes desta chamada, ocorre a invocação ao método `RemoverTodosProdutos()`, que retira da nota de entrada todos os produtos registrados nela. É preciso estar atento para o caso de ocorrência de algum erro na execução do método do `Repository`, pois quando a nota for realmente removida da coleção, seus produtos já não existirão mais.

Resolução 4.44: método Remove() da classe NotaEntradaController

```
public void RemoveNotaEntrada(NotaEntrada notaEntrada) {
    this.notasEntrada.Remove(notaEntrada);
}
```

A última implementação referente à interação com os dados que pertencem ao corpo da nota é apresentada na sequência. Ela se refere ao evento `SelectionChanged` do `DataGridView` que exibe as notas de entradas registradas.

Resolução 4.45: método SelectionChanged() do DataGridView de Notas de Entrada

```
private void dgvNotasEntrada_SelectionChanged(
        object sender, EventArgs e) {
    try {
        this.notaAtual = this.controller.
            GetNotaEntradaById((Guid) dgvNotasEntrada.
            CurrentRow.Cells[0].Value);
        txtIDNotaEntrada.Text = notaAtual.Id.
            ToString();
        txtNumero.Text = notaAtual.Numero;
        cbxFornecedor.SelectedItem = notaAtual.
            FornecedorNota;
        dtpEmissao.Value = notaAtual.DataEmissao;
        dtpEntrada.Value = notaAtual.DataEntrada;
        UpdateProdutosGrid();
    }
    catch (Exception exception) {
        this.notaAtual = new NotaEntrada();
    }
}
```

O método implementado está dentro de um bloco `try...catch()`. Esta técnica foi usada, pois o método é chamado a qualquer alteração de estado da fonte de dados e, em caso de inexistência de dados, uma exceção pode ocorrer.

O método que realiza a busca pelo registro selecionado no `DataGridView` é o `GetNotaEntradaById()`, implementado no controlador – visto a seguir. Este retorna a nota encontrada, que é armazenada, e então tem suas propriedades utilizadas para popular os controles do formulário. Como cada nota possui um conjunto de produtos armazenados, o método `UpdateProdutosGrid()` – **Resolução 4.48** – é invocado para atualizar o `DataGridView` de produtos.

Resolução 4.46: método GetNotaEntradaById() da classe NotaEntradaController

```
public NotaEntrada GetNotaEntradaById(Guid Id) {
    return this.repository.GetNotaEntradaById(Id);
}
```

Na sequência, mostro o método `GetNotaEntradaById()` da classe `Repository`.

Resolução 4.47: método GetNotaEntradaById() da classe Repository

```
public NotaEntrada GetNotaEntradaById(Guid Id) {
    var notaEntrada = this.notasEntrada[
    this.notasEntrada.IndexOf(
        new NotaEntrada() {Id = Id}
    )];
    return notaEntrada;
}
```

A busca pela nota de entrada que possua o `Id` enviado para o método se dá por meio do método `IndexOf()` da coleção, o que pode variar de acordo com a coleção a ser escolhida.

O método `IndexOf()` faz uma pesquisa na coleção de objetos à procura de um item que corresponda ao enviado como argumento. Encontrando-o, é retornado o índice desse elemento, tendo como base o comportamento de uma matriz.

Resolução 4.48: método UpdateProdutosGrid()

```
private void UpdateProdutosGrid() {
    dgvProdutos.DataSource = null;
    if (this.notaAtual.Produtos.Count > 0) {
        dgvProdutos.DataSource = this.notaAtual.
            Produtos;
    }
}
```

4.11 Implementando a interação do usuário com o registro dos produtos da Nota de Compra

Alguns cuidados devem ser previstos nessa interface criada. Um deles é que o usuário não poderá registrar um produto se não existir uma nota registrada e selecionada. Para isso, todos os controles que permitem interação devem estar desabilitados, e ser apenas habilitados quando um novo registro for solicitado e o requisito de uma nota de entrada existir e for cumprido.

Altere para `False` a propriedade `Enabled` dos controles: `cbxProduto`, `txtCusto`, `txtQuantidade`, `btnGravarProduto`, `btnCancelarProduto` e `btnRemoverProduto`. Os nomes apresentados são autoexplicativos para identificar a qual controle se referem.

Caso ainda não tenha nomeado os controles, faça-o agora. O controle de `ID` será sempre desabilitado, pois não permite interação. A habilitação desses controles se dará no momento em que o usuário clicar no botão `Novo` produto, desde que a validação seja bem sucedida. A seguir, apresento o código para o método que captura o evento `Click` do botão `Novo produto`.

Resolução 4.49: método que captura o evento Click do botão para um novo produto

```
private void BtnNovoProduto_Click(object sender,
        EventArgs e) {
    ClearControlsProduto();
    if (txtIDNotaEntrada.Text == string.Empty) {
        MessageBox.Show("Selecione a NOTA, que terá "+
            "inserção de produtos, no GRID");
    }
    else {
        ChangeStatusOfControls(true);
    }
}
```

Verifica-se que a contradição à expressão avaliada (o `else`) causa a chamada a um método em específico, o `ChangeStatusOfControls()`, man-

dando como argumento o valor booleano `true`. A implementação desse método pode ser verificada na sequência.

Resolução 4.50: método responsável pela habilitação e desabilitação de controles referentes aos produtos da nota

```
private void ChangeStatusOfControls(bool newStatus) {
    cbxProduto.Enabled = newStatus;
    txtCusto.Enabled = newStatus;
    txtQuantidade.Enabled = newStatus;
    BtnNovoProduto.Enabled = !newStatus;
    BtnGravarProduto.Enabled = newStatus;
    BtnCancelarProduto.Enabled = newStatus;
    BtnRemoverProduto.Enabled = newStatus;
}
```

Com o usuário clicando no botão `Novo` para inserir um produto, os controles são todos habilitados e o botão `Novo` torna-se inativo. Com a interface liberada para o usuário informar os dados, ele pode optar em `Gravar` sua digitação ou `Cancelar` qualquer informação digitada nos controles. A seguir, apresento a implementação para o método que captura o evento `Click` do botão `Gravar produto`.

Resolução 4.51: método que captura o evento Click do botão Gravar produto

```
private void BtnGravarProduto_Click(object sender,
        EventArgs e) {
    var produtoNota = new ProdutoNotaEntrada() {
        Id = (txtIDProduto.Text == string.Empty ?
            Guid.NewGuid() : new Guid(txtIDProduto.
            Text)),
        PrecoCustoCompra = Convert.ToDouble(
            txtCusto.Text),
        ProdutoNota = (Produto) cbxProduto.
            SelectedItem,
        QuantidadeComprada = Convert.ToDouble(
            txtQuantidade.Text)
    };
```

```
    this.notaAtual.RegistrarProduto(produtoNota);
    this.notaAtual = this.controller.Update(
        this.notaAtual);
    ChangeStatusOfControls(false);
    UpdateProdutosGrid();
    ClearControlsProduto();
}
```

A primeira instrução a ser discutida é a chamada ao método `RegistrarProduto()`, pertencente ao objeto `notaAtual`, que pode ser verificado na sequência.

Resolução 4.52: método RegistrarProduto() da classe FormNotaEntrada

```
public void RegistrarProduto(ProdutoNotaEntrada produto) {
    if (this.Produtos.Contains(produto))
        this.Produtos.Remove(produto);
    this.Produtos.Add(produto);
}
```

O método `RegistrarProduto()` inicialmente busca pelo produto recebido e, caso ele já exista na coleção, ele é removido e o novo inserido. Esta técnica é utilizada para conhecimento de meios e usos de métodos disponíveis pela coleção adotada. É possível evitar esta situação escolhendo outras coleções com mais recursos já implementados.

O método `Cancelar edição/inserção de produtos` tem sua implementação apresentada a seguir.

Resolução 4.53: Método que captura o evento Click do botão Cancelar edição/inserção de produto

```
private void BtnCancelarProduto_Click(object sender,
        EventArgs e) {
    ClearControlsProduto();
    ChangeStatusOfControls(false);
}
```

O último botão, `Remover` produto, tem seu comportamento no método representado adiante.

Resolução 4.53: método que captura o evento Click do botão Cancelar edição/inserção de produto

```
private void BtnRemoverProduto_Click(object sender,
        EventArgs e) {
    this.notaAtual.RemoverProduto(
        new ProdutoNotaEntrada() {
            Id = new Guid(txtIDProduto.Text)
        }
    );
    this.controller.Update(this.notaAtual);
    UpdateProdutosGrid();
    ClearControlsProduto();
    ChangeStatusOfControls(false);
}
```

Note que a remoção do produto da nota se faz exatamente pela retirada dele da coleção de produtos e, em seguida, o método `Update()` do controlador é invocado para atualizar a nota.

4.12 Conclusão

Este capítulo buscou apresentar técnicas para associar objetos, criando uma aplicação com diversos projetos e aplicando o conceito de MVC com um repositório baseado em coleções. Durante o desenvolvimento do projeto de exemplo, diversas observações e técnicas foram apresentadas e discorridas.

Ao final, obteve-se uma aplicação com as operações CRUD (*Create, Retrieve, Update and Delete*) para um conjunto de dados *master-detail*. Aos poucos, o arcabouço de ferramentas e técnicas de desenvolvimento, recursos da linguagem e ambientação ao IDE vem aumentando. Com o acesso a banco de dados, no próximo capítulo já será possível o desenvolvimento de aplicações realmente funcionais e condizentes com as necessidades de uma empresa.

Capítulo 5

Acesso a dados por meio do ADO.NET

Toda aplicação desenvolvida manipula e processa dados. Esses dados podem ser transientes – informados diretamente na aplicação e imediatamente processados –, podem ser obtidos de fontes externas – como um arquivo texto – ou ainda registrados em uma coleção e depois armazenados em um arquivo. Entretanto, o meio mais comum para persistência e obtenção de dados é o uso de Sistemas Gerenciadores de Base de Dados (SGBD), no qual tabelas são criadas para receber e/ou fornecer dados da/para aplicação. Este capítulo apresenta o uso do ADO.NET para realização desta atividade, onde os exemplos demonstrarão o que esse framework oferece.

Inicialmente, será trabalhado apenas com código, para conhecimento da API (*Application Program Interface*), e depois serão também apresentados re-

cursos oferecidos pela plataforma e o Visual Studio. Procure fazer uso das técnicas apresentadas nos capítulos anteriores em relação à formatação visual e à técnica de diversos projetos em uma solução, aplicando o MVC.

5.1 Introdução ao ADO.NET

O ADO.NET é um conjunto de classes (API) que possibilita o acesso e a manipulação de diversos tipos de fontes de dados (*data sources*) para programadores do .NET Framework. Normalmente, uma fonte de dados é uma base de dados, e suas tabelas são visões e *stored procedures*, mas também podem ser um arquivo de texto, uma planilha do Excel, um arquivo XML ou ainda um *web service*.

Em relação às bases de dados, que são o foco deste capítulo, a comunicação com os SGBDs é possível com qualquer um dos mais utilizados, como: SQL Server – que é o usado no livro –, MySQL, Firebird, Oracle etc.

Os componentes considerados como os pilares do ADO.NET incluem os objetos:

a) `Connection`, responsável por efetuar a conexão com o banco de dados;

b) `Command`, responsável por executar comandos diretamente no banco de dados;

c) `DataAdapter`, utilizado para preencher um objeto `DataSet`.

Esses três itens são conhecidos como **.NET Framework Data Provider** e, além deles, o `DataSet` também é uma parte fundamental. Todos esses componentes serão apresentados e trabalhados.

É muito comum uma confusão entre os termos **banco de dados** e **base de dados**. Em alguns casos, quem lê (ou escreve) nem se depara com essa diferença, mas é extremamente importante que você a saiba. Quando se usa o termo "banco de dados", busca-se referenciar o software, o produto com o qual a aplicação se conectará. Já o termo "base de dados" refere-se ao produto fornecido pelos Bancos de Dados, ou seja, os registros neles armazenados, quer sejam estes processados ou não.

5.2 Criando a base de dados utilizando o Visual Studio

Para que os recursos do ADO.NET possam ser apresentados, exemplificados e aplicados, faz-se necessário uma base de dados. Inicialmente, a manipulação dessa base para criação de tabelas será realizada por meio do Visual Studio. Para isso, crie uma nova solução, e nela um projeto `Windows Forms`. Nesse projeto, crie uma pasta chamada `App_Data`.

A figura a seguir apresenta a `Solution Explorer` obtida:

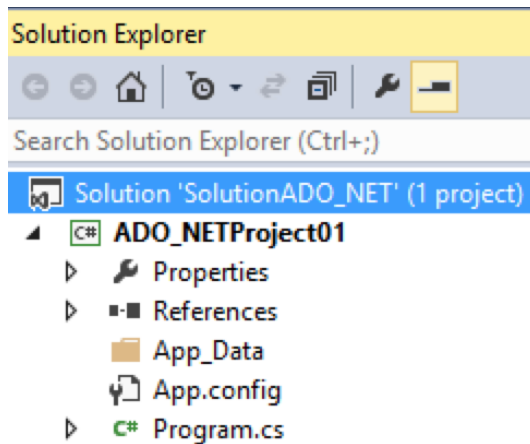



Fig. 5.1: Selecionando projeto para ter referência adicionada

Clique com o botão direito do mouse na nova pasta criada (`App_Data`) e selecione no menu a opção `Add` → `New Item`. Na janela que se exibe, selecione a categoria `Data` e, dentro dela, o template `Service-based Database`. Dê o nome `ADO_NETDataBase.mdf` ao arquivo de base de dados que será criado. Veja a figura:

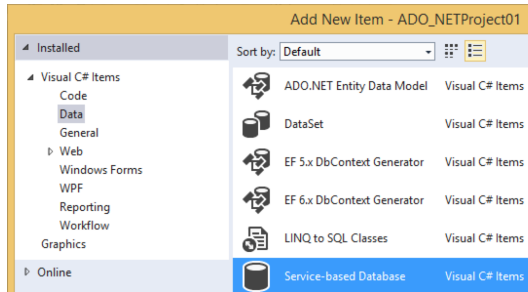



Fig. 5.2: Criando a base de dados

Um `Service-base Database` é um arquivo que representa uma base de dados. É nesse arquivo que estarão os objetos criados e usados pela aplicação, como: tabelas, *indexes*, *constraints*, *stored procedures*, *functions* etc.

Após confirmar a criação da base de dados, caso o erro apresentado pela figura 5.3 apareça, é necessário que o serviço que representa o SQL Server seja iniciado.

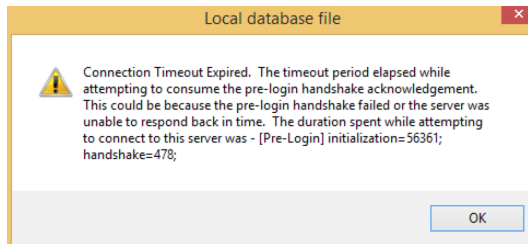



Fig. 5.3: Erro referente ao SQL Server não estar iniciado na criação da base de dados

Para iniciar o serviço do SQL Server, acesse os `Serviços Locais do Windows`. Lá, localize o serviço, clique com o botão direito sobre ele e na opção `Iniciar`. Caso essa opção esteja desabilitada, é porque na criação da base de dados esse procedimento estava em andamento e não havia concluído, bastando repetir o procedimento já explicado para a criação do arquivo. Veja na figura a seguir o serviço sendo iniciado:

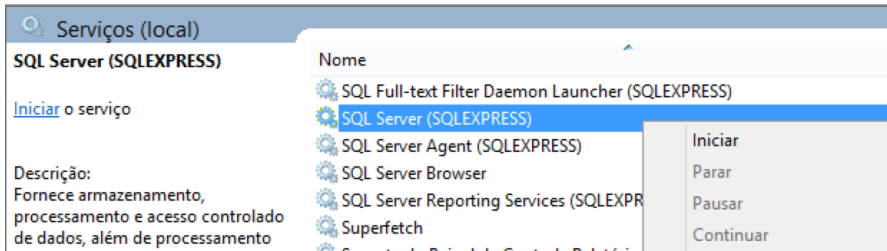



Fig. 5.4: Iniciando o serviço local do SQL Server

Após a criação ter ocorrido com sucesso, o arquivo criado é exibido na `Solution Explorer`, como pode ser verificado na figura:

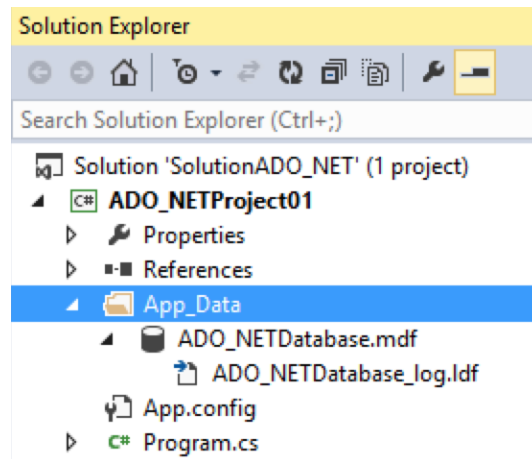



Fig. 5.5: Solution Explorer com o arquivo de base de dados criado

Com a base de dados criada, é preciso agora fazer as tabelas que a vão compor. Para essa primeira atividade, seguindo o exemplo do capítulo anterior, será criada inicialmente a tabela referente aos fornecedores. Para isso, acesse a `Server Explorer` (menu `View → Server Explorer`), como mostra a figura 5.6. Se, ao expandir o nó `Data Connections`, a conexão com a base criada aparecer com um ícone em vermelho, significa que ela não foi aberta. Assim, basta expandir o nó da base de dados para a conexão ser

estabelecida e exibir as categorias de objetos disponíveis.

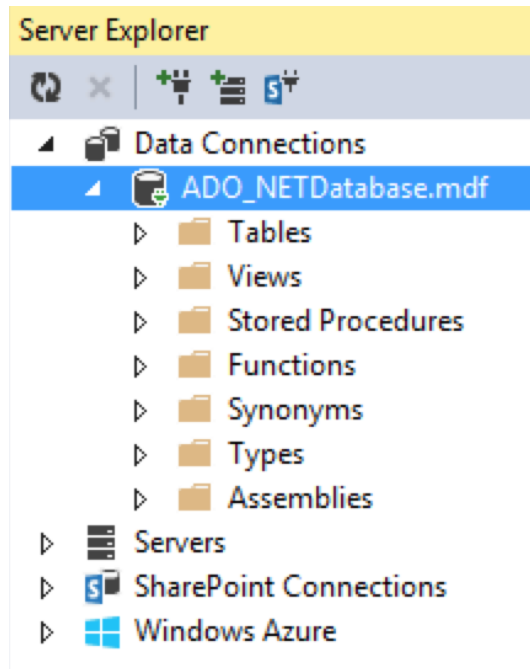



Fig. 5.6: Server Explorer exibindo a base de dados criada

Com as categorias de possíveis objetos para a base de dados, é preciso criar a tabela que conterá e fornecerá os dados referentes aos fornecedores. Para isso, clique com o botão direito do mouse sobre `Tables` e depois na opção `Add New Table`:

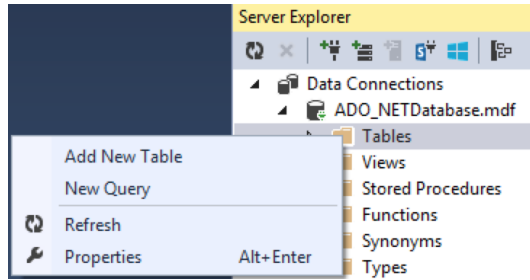



Fig. 5.7: Adicionando uma tabela à base de dados

Caso o menu de contexto exibido em sua máquina seja diferente do apresentado na figura 5.7, aparecendo apenas as opções `Refresh` e `Properties`, a ferramenta **SQL Server Data Tools** (SSDT) não está instalada em seu ambiente. No Visual Studio 2013, ela é instalada automaticamente com o IDE. Mas, se mesmo assim as opções não aparecerem, algum erro ocorreu durante a instalação e ela precisa ser realizada novamente. Para o Visual Studio 2012, é preciso buscar o download dessa ferramenta.

O **SSDT** é uma ferramenta que permite que a criação e manipulação de uma base de dados possa ser realizada diretamente no Visual Studio, sem a necessidade de uma ferramenta separada, como o **SQL Server Management Studio**.

Após a confirmação de adição de uma nova tabela, uma janela oferecida pelo **SSDT** é exibida. É por meio dela que a tabela poderá ser criada tanto visualmente como por meio de instruções SQL:

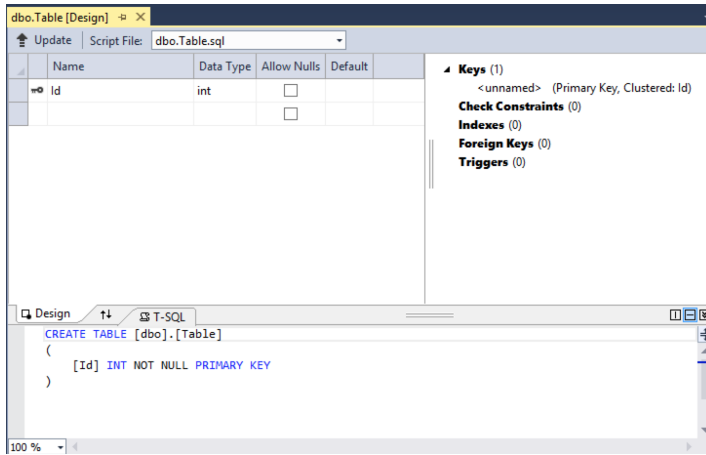



Fig. 5.8: Ambiente para definição de estrutura de tabelas por meio do SQL Server Data Tools no Visual Studio

No contexto apresentado neste livro, em relação aos bancos de dados relacionais, uma **tabela** (*table*) pode ser definida como um conjunto de dados dispostos em um número finito de colunas (campos) e ilimitado de linhas (registros ou tuplas).

As **colunas** são consideradas como **campos** na tabela. Elas caracterizam os dados que comporão a informação esperada por cada linha/registro ou tupla. Cada coluna possui um tipo de dado específico.

As **linhas** (**registros** ou **tuplas**) podem trazer todas as informações armazenadas em uma tabela, com todos os campos ou alguns deles. É possível também que tabelas sejam combinadas para retornarem registros que gerem informação composta. Por exemplo, em uma única linha, é possível ter dados das quatro tabelas que serão processadas neste capítulo: fornecedor, produto, notas de compra e dados de cada produto comprado. A forma de referenciar inequivocamente uma única linha é por meio da utilização de uma chave primária.

Os registros de uma tabela, na maioria das vezes, não podem ser idênticos. Para este requisito ser garantido, é preciso que a tabela tenha sofrido um processo de **normalização**. Este tema é extenso e foge dos objetivos deste

livro. Desta maneira, para simplificar, é adotado o conceito de **chave primária** (ou **primary key**) e de **identidade** (ou **autoincremento**), em que o valor para a chave primária é criado automaticamente e não faz parte dos domínios naturais de cada tabela.

As colunas (campos) da tabela podem ser informadas tanto na área de desenho, onde aparece um grid com a definição predefinida para `Id` – que será a chave primária para a tabela –, quanto na parte onde aparece o código SQL. Nesta parte do código, mude o nome da tabela para `Fornecedores` e, na parte visual, selecione a linha da coluna `Id`. Em seguida, ative a janela de propriedades, pressionando a tecla `F4`. Altere a propriedade de configuração da coluna como identidade. O resultado pode ser visto na figura:

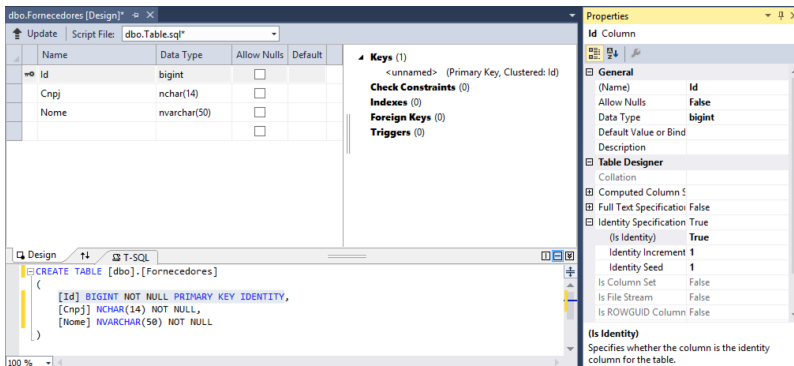



Fig. 5.9: Estrutura para a tabela Fornecedores

SQL (*Structured Query Language* – Linguagem de Consulta Estruturada) é a linguagem para banco de dados relacional. Muitas das características originais do SQL foram inspiradas na álgebra relacional. Por ser um assunto que por si só merece um livro, esse tema não será detalhado além do necessário neste livro.

Quando se trabalha com SGBDs, um conjunto de códigos está disponível, seja para definição da base de dados, como para obtenção de dados e informações dessas bases de dados. A instrução SQL `CREATE TABLE` é uma instrução para definição da base de dados, uma **DDL** (*Data Definition Language*).

Com a definição da estrutura pronta, é preciso confirmar a criação da tabela. Para isso, no topo da janela, clique no botão `Update`. Essa operação será responsável por apresentar as operações que serão realizadas na atualização da base de dados:

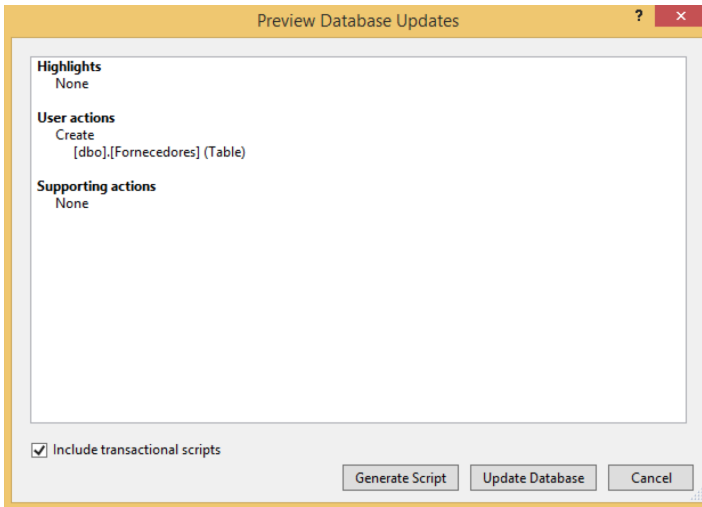



Fig. 5.10: Janela de pré-atualização da base de dados

Para realizar a atualização da base de dados, clique no botão `Update Database`. Se tudo ocorrer bem, ao final do processamento, a console do `Data Tools Operation` exibirá o resultado mostrado na figura a seguir:

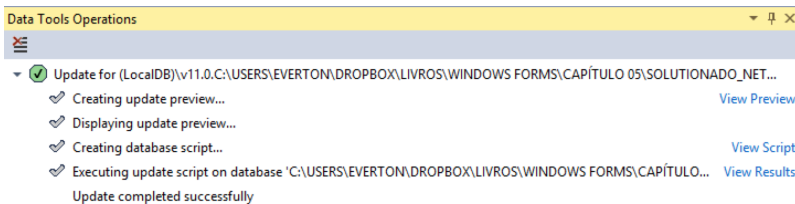



Fig. 5.11: Console do Data Tools Operation após a atualização da base de dados

Para verificar a tabela criada no `Server Explorer`, pressione o botão

de `Refresh`. Ela poderá ser visualizada, de acordo com a figura 5.12.

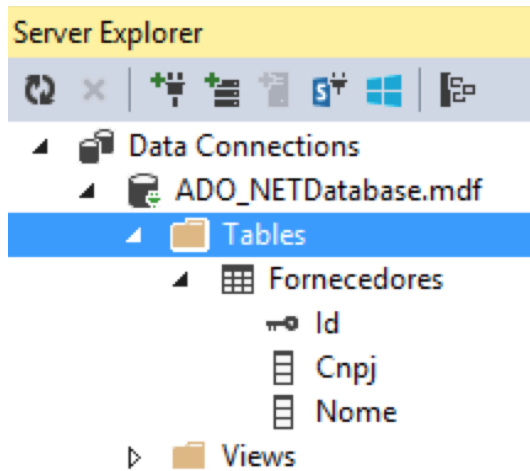



Fig. 5.12: Server Explorer exibindo a nova tabela criada e suas colunas/campos

5.3 Realizando operações relacionadas ao CRUD em uma tabela de dados

Inicialmente, o formulário para interação com o usuário, para a manutenção dos dados de fornecedores, será semelhante ao trabalhado no capítulo 4:

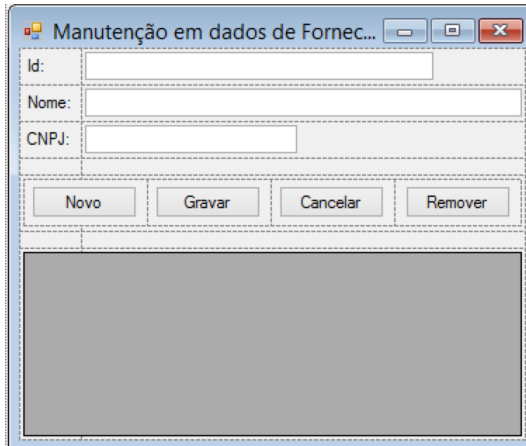



Fig. 5.13: Formulário para manutenção nos dados de Fornecedores

Após o desenho do formulário, é preciso fornecer à aplicação informações para que ela possa acessar o arquivo de base de dados criado. Esta configuração deve ser feita no arquivo `App.Config`, na raiz (*root*) do projeto. O nome para essa configuração de acesso a uma base de dados é `Connection String`. A seguir, trago o código necessário para a configuração de uma string de conexão com o arquivo de base de dados criado anteriormente.

Resolução 5.1: connectionString para conexão com o banco criado

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0"
            sku=".NETFramework,Version=v4.5" />
    </startup>
    <connectionStrings>
        <add name="CS_ADO_NET"
            connectionString="Data Source=(LocalDB)\v11.0;
            AttachDbFilename=C:\Users\Everton\Dropbox\
            Livros\Windows Forms\Capítulo 05\SolutionADO_NET
            \ADO_NETProject01\App_Data\ADO_NETDatabase.mdf;
```

```
            Integrated Security=True"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

Toda a configuração necessária para uma aplicação `Windows Forms` pode ser realizada no arquivo `App.Config`, além de usar implementações em C#, que possam configurar a aplicação também em tempo de execução. A configuração apresentada está (e deve ser sempre) com base em XML. O elemento mais externo visualizado é o `<configuration>`, e a configuração de uma `connectionString` deve ser realizada dentro desse elemento, por meio de elementos `<connectionString>`.

No contexto trabalhado neste livro, que é o acesso a dados, uma *connection string* (ou string de conexão) é uma string que especifica informações sobre uma fonte de dados e como acessá-la. Ela passa, por meio de código, informações para um `driver` ou `provider` sobre o que é necessário para iniciar uma conexão. Normalmente, a conexão é para uma base de dados, mas também pode ser usada para uma planilha eletrônica ou um arquivo de texto, dentre outros. Uma connection string pode ter atributos como nome do driver, servidor e base de dados, além de informações de segurança, como nome de usuário e senha.

Dentro do elemento `<connectionString>`, há um elemento `<add>`, que adiciona ao contexto da aplicação uma nova connection string, e alguns atributos são definidos:

a) `name` define o nome para a conexão a ser adicionada, neste caso `CS_ADO_NET`;

b) `connectionString` é um atributo complexo, onde:

- `Data Source` é o servidor onde o banco de dados está e, neste caso, é apontado o `Local Data Base`. O valor `(LocalDB)\v11.0` refere-se ao caminho no qual o servidor local de banco de dados está instalado que, por padrão, é `C:\Program Files\Microsoft SQL Server`\LocalDB\Binn`.

- `AttachDbFileName` refere-se, para o banco de dados local, ao caminho físico para o arquivo que representa a base de dados. Para obter o caminho físico do arquivo de base de dados, é preciso selecionar o arquivo na `Solution Explorer` e, então, na janela de propriedades, obter o valor de `Full Path` (figura 5.14).
- `Integrated Security` define como a autenticação será utilizada na conexão. Quando recebe `True`, assume-se a autenticação do sistema operacional (Windows, no caso). Já se o valor atribuído for `False`, será necessário informar o número de usuário e senha.

c) `providerName` fornece para a conexão o nome do `Data Provider` responsável por realizar a conexão com a base de dados.

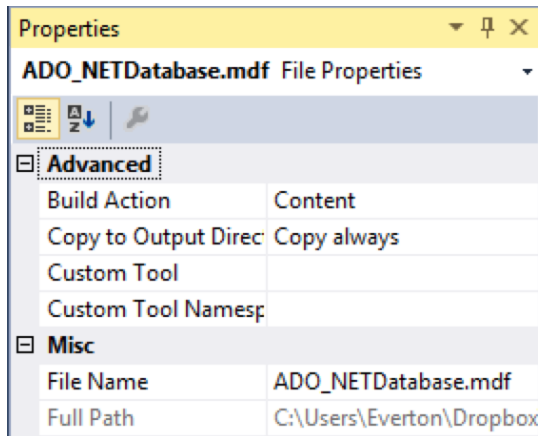



Fig. 5.14: Obtendo o caminho físico do arquivo de base de dados

A escrita de uma *connection string* pode variar de acordo com o modo de acesso ao banco de dados e também com o banco de dados. Para auxiliar nesta atividade, cada produto oferece informação de como criar uma string de conexão, precisando apenas recorrer à documentação disponibilizada. Buscando minimizar a dificuldade, existe um site que é referência no assunto, o http://www.connectionstrings.com, que fornece tutoriais, dicas e artigos relevantes.

5.4 Inserindo registros na tabela de fornecedores

Com o apresentado anteriormente, relacionado ao desenho da janela, na configuração para conexão com a base de dados, já se tem o necessário para a primeira operação relacionada a um CRUD, que é a inserção de um novo registro.

Crie um método para o evento `Click` do botão `Gravar`, realizando um duplo clique sobre o botão. Você pode também fazer isso selecionando o botão e, na `Properties View` em `Events`, selecionar o `Click` e dar um duplo clique na área em branco, ao lado de seu nome.

Implemente no corpo do método o código apresentado na sequência.

Resolução 5.2: método que implementa o evento Click do botão Gravar (Versão 1)

```
private void btnGravar_Click(object sender, EventArgs e) {
    string connectionString = ConfigurationManager.
        ConnectionStrings["CS_ADO_NET"].ConnectionString;
    SqlConnection connection = new SqlConnection(
        connectionString);
    connection.Open();
    SqlCommand command = connection.CreateCommand();
    command.CommandText = "insert into FORNECEDORES(nome,
        cnpj) values(@nome, @cnpj)";
    command.Parameters.AddWithValue("@nome", txtNome.Text);
    command.Parameters.AddWithValue("@cnpj", txtCNPJ.Text);
    command.ExecuteNonQuery();
    connection.Close();
    MessageBox.Show("Fornecedor registrado com sucesso");
}
```

É possível verificar na primeira linha do método a declaração e inicialização de uma variável do tipo `string`, que receberá a connection string definida como `CS_ADO_NET` no `App.Config`. A obtenção desse valor é realizada por meio da propriedade `ConnectionStrings`, da classe `ConfigurationManager`. Note que o nome da connection string é inserido

como `string` dentro dos colchetes e, após eles, é informada a propriedade `ConnectionString`.

A classe `ConfigurationManager` permite o acesso por meio de código, em tempo de execução, ao arquivo de configuração (`App.Config`) e suas seções. Uma de suas funcionalidades é o acesso rápido às seções `<appSettings>` e `<connectionStrings>`. A propriedade `ConnectionStrings` fornece os dados da seção `<connectionStrings>` do arquivo de configuração da aplicação.

Caso a classe `ConfigurationManager` apareça como desconhecida (ou inexistente) em seu código, adicione ao bloco de `namespaces`, no início do arquivo, a cláusula `using System.Configuration`. Se mesmo assim o problema persistir, é preciso adicionar ao seu projeto o acesso ao `Assembly` dessa classe. Para isso, na `Solution Explorer`, clique com o botão direito do mouse sobre `References` e depois em `Add Reference`. Na janela que é exibida, no lado esquerdo clique na categoria `Framework` e no quadro central procure por `System.Configuration 4.0.0.0`. Então, marque a caixa de seleção, para que o arquivo seja adicionado a seu projeto:

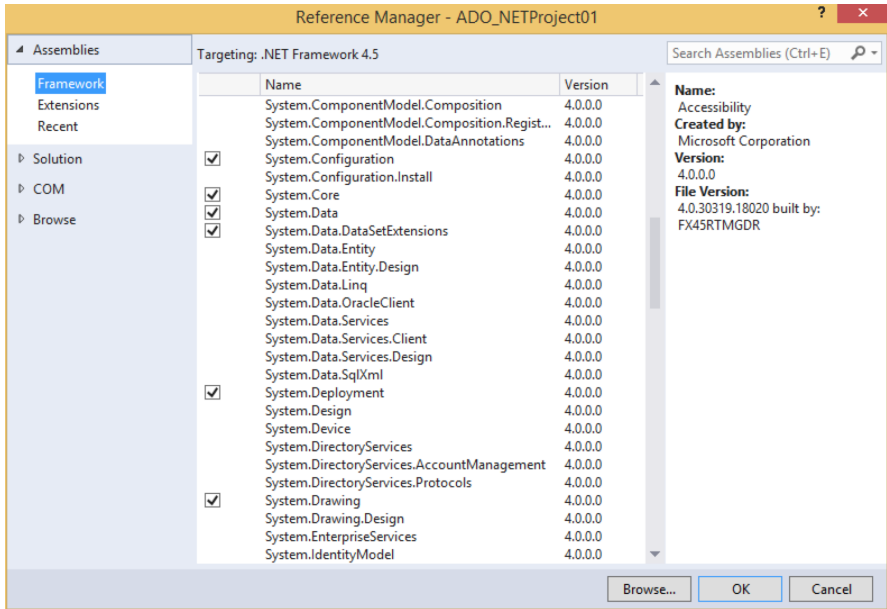



Fig. 5.15: Adicionando um assembly às referências do projeto

A segunda instrução implementada no método refere-se à criação do objeto responsável pela conexão com o banco SQL Server, que é um objeto da classe `SqlConnection`. Na chamada ao construtor da classe, em sua instanciação, é enviada uma string, que é a connection string anteriormente recuperada. Após o objeto ter sido instanciado, a instrução seguinte abre a conexão, por meio do método `Open()`.

A classe `SqlConnection` é utilizada para gerenciar conexões com bases de dados do SQL Server. Por meio de objetos dessa classe, as conexões são abertas e fechadas, e permitem a criação de objetos que representam comandos/instruções a serem executados na base de dados. Ela fornece um grande número de recursos que, conforme forem sendo usados, serão apresentados.

Com a conexão com a base de dados estabelecida, é preciso criar um objeto que possibilite a execução de instruções na base de dados. Esses objetos são da classe `SqlCommand` e são obtidos de acordo com a instrução da quarta linha da implementação no método.

A classe `SqlCommand` é responsável por executar instruções SQL em

bancos de dados SQL Server, podendo ser usada tanto para consultas como para instruções "não query", como *updates*, *inserts* e execução de *procedures*.

Dentre as diversas propriedades oferecidas pela classe, as utilizadas na implementação são:

- `CommandText`, que representa a instrução SQL a ser executada, como um `select`, `update`, `insert` ou o nome de um *stored procedure* existente;

- `Parameters`, que é uma coleção de objetos do tipo `SqlParameter` e, assim como a maioria das coleções, possui métodos como `Add()` e `Remove()`, dentre outros.

Os parâmetros dessa lista devem corresponder àqueles definidos no `CommandText` (identificados por `@` antes do nome). Os objetos `SqlParameter` possuem diversos métodos também. Um deles, que é usado na implementação, é o `AddWithValue()`, cujo uso é recomendado em vez de se utilizar apenas o `Add()`.

Retornando à classe `SqlCommand`, ela também possui métodos. Dentre eles, há o `ExecuteNonQuery()`, responsável por executar as instruções de atualização de dados, como no caso o `insert`. Esse método retorna o número de linhas que sofreram atualizações pela sua execução.

As instruções SQL, como o `insert` apresentado, fazem parte da categoria **DML** (*Data Manipulation Language*). Essas instruções retornam dados (`select`) e os atualizam (`insert`, `update` e `delete`).

Finalizando a implementação do método, existe a chamada ao método `Close()` do objeto `SqlConnection` e uma exibição ao usuário de uma mensagem referente ao sucesso da operação.

Da mesma maneira que, no exemplo do capítulo 4 na **Resolução 4.19** (seção 4.7), os campos eram limpos, é preciso fazer o mesmo após a inserção do fornecedor na tabela. Siga o exemplo apresentado nesta parte para implementar essa funcionalidade. Abra uma nova instância do Visual Studio com a solução aberta para lhe auxiliar, pois, além dessa implementação, outras já realizadas serão apontadas.

5.5 Obtendo todos os fornecedores existentes na base de dados

Para obter todos os fornecedores existentes na tabela e popular o `DataGridView` com estes dados, é preciso realizar uma consulta na tabela da base de dados. Esta é realizada por meio da instrução SQL `select`.

Ao contrário do realizado na implementação da inserção de um registro que está atrelada a um método que captura um evento, essa nova implementação será realizada em um método específico, seguindo a proposta deste livro, de crescimento do conhecimento a cada implementação apresentada.

Resolução 5.3: método responsável por obter todos os fornecedores registrados na tabela de dados (Versão 1)

```
private void GetAllFornecedores() {
    string connectionString = ConfigurationManager.
    ConnectionStrings["CS_ADO_NET"].ConnectionString;
    var connection = new SqlConnection(connection);

    var adapter = new SqlDataAdapter("select id, cnpj, nome
        from FORNECEDORES", connectionString);
    var builder = new SqlCommandBuilder(adapter);

    var table = new DataTable();
    adapter.Fill(table);

    dgvFornecedores.DataSource = table;
    connection.Close();
}
```

Na implementação do método `GetAllFornecedores()`, repete-se a declaração da `connectionString` da obtenção e encerramento da conexão com o SQL Server, o que já deve levar a uma reflexão de refatoração dos dois métodos já implementados, buscando uma reutilização.

Na declaração das variáveis `adapter`, `builder` e `table` já foi utilizada a maneira mais atual de declaração de variáveis, como visto em capítulos anteriores. Note que a conexão não foi aberta de maneira explícita, pois o método

`Fill()` do `adapter` tem como uma de suas atividades a abertura da conexão nele registrada.

O processo para a seleção dos dados se dá por meio da criação de um `adapter`, que possui a instrução SQL a ser executada no servidor e a conexão com informações da base de dados onde ela será executada. Com o `adapter` criado, é preciso gerar os `commands` para as instruções SQL informadas nele. Diferentemente do apresentado na inserção dos registros, onde foi usada a classe `SqlCommand`, essa implementação usa a classe `SqlCommandBuilder`, que gera o `SqlCommand` com base na string SQL enviada em seu construtor.

Na sequência, um `table` é criado e, então, preenchido com os dados obtidos por meio da execução do método `Fill()` do `adapter`, que recebe a `table` que será preenchida. Finalizando, o `DataGridView` recebe a `table` já populada e a conexão é, então, fechada.

A classe `SqlDataAdapter` representa uma "ponte" entre a base de dados e o `DataTable` (ou `DataSet`). O método `Fill()` usado na implementação excuta a instrução SQL `select` e utiliza o seu retorno para popular o `DataTable` enviado como parâmetro, além de também realizar a abertura de sua conexão.

A classe `SqlCommandBuilder` gera `Commands` para serem utilizados por um `SqlDataAdapter`.

A classe `DataTable` representa, de maneira lógica, uma tabela. Esta pode ser uma tabela física – existente em uma base de dados –, ou apenas uma tabela criada e utilizada em tempo de execução. A classe oferece diversas propriedades e métodos que podem ser usados de acordo com as necessidades.

Para visualizar na interface com o usuário a execução desse método, é preciso realizar algumas alterações. A primeira é no construtor, para que, ao exibir o formulário, os fornecedores registrados já possam ser visualizados. A segunda é no método no qual é realizada a inserção dos dados informados pelo usuário, na janela de manutenção em dados de fornecedores. Veja ambas resoluções na sequência.

Resolução 5.4: método responsável por obter todos os fornecedores registrados na tabela de dados (Versão 1)

```
public FornecedoresCRUD() {
    InitializeComponent();
    GetAllFornecedores();
}
```

Resolução 5.5: método que implementa o evento Click do botão Gravar (Versão 2)

```
private void btnGravar_Click(object sender, EventArgs e) {
    using (var connection = new SqlConnection(
    this.connectionString)) {
        connection.Open();
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "insert into FORNECEDORES(
            nome, cnpj) values(@nome, @cnpj)";
        command.Parameters.AddWithValue("@nome",
            txtNome.Text);
        command.Parameters.AddWithValue("@cnpj",
            txtCNPJ.Text);
        command.ExecuteNonQuery();
    }
    MessageBox.Show("Fornecedor registrado com sucesso");
    ClearControls();
    GetAllFornecedores();
}
```

A primeira alteração que pode ser notada é a retirada da definição da variável `connectionString`, que não faz mais parte do método, mas sim da classe (**Resolução 5.6**). A segunda alteração refere-se ao uso da instrução `using` na declaração e inicialização da variável `connection`. Esta é a maneira mais correta de se implementar quando se utiliza de recursos que precisam ser fechados e liberados.

Na Versão 1 da **Resolução 5.5**, não havia sido inserida a chamada ao método `Dispose()` de `connection` após a chamada ao método `Close()`. Essa chamada seria necessária caso não fosse usado o `using`. Também não havia sido inserida a chamada ao método `ClearControls()`, que tem a mesma finalidade do método de mesmo nome utilizado no capítulo 4 e que pode ser visualizado na **Resolução 5.7**.

Finalizando, a última instrução realiza a chamada ao método GetAllFornecedores(), apresentado anteriormente na **Resolução 5.3**.

Resolução 5.6: declarações de membros da classe (Versão 1)

```
public partial class FornecedoresCRUD : Form {
    private string connectionString = ConfigurationManager.
    ConnectionStrings["CS_ADO_NET"].ConnectionString;
    //Código complementar omitido
```

Resolução 5.7: método ClearControls() (Versão 1)

```
private void ClearControls() {
    txtID.Text = string.Empty;
    txtNome.Text = string.Empty;
    txtCNPJ.Text = string.Empty;
    txtNome.Focus();
}
```

Quando se utiliza um objeto que encapsula qualquer recurso, é preciso garantir que quando não precisarmos mais desse objeto seu método Dispose() seja invocado. Esta atividade pode ser mais facilmente realizada com a instrução using (*using statement*).

A instrução using simplifica o código que cria, finaliza e limpa o objeto. Ela obtém o recurso especificado, executa as instruções e finalmente invoca o método Dispose() do objeto, para limpá-lo.

5.6 Obtendo um fornecedor específico na base de dados

Para que seja possível ao usuário alterar os dados de um fornecedor, ou ainda removê-los da base de dados, é preciso – nesta aplicação exemplo – que ele seja selecionado no DataGridView e tenha seus dados exibidos. Para que os dados do fornecedor selecionado possam ser exibidos, é preciso que o for-

necedor tenha sido buscado na tabela e, então, um objeto que o represente seja populado, para assim fornecer os dados para os componentes visuais.

A classe que representará um registro da tabela de fornecedores pode ser visualizada na sequência, que é semelhante à classe de mesmo nome implementada no capítulo 4, mudando apenas o tipo de dado da propriedade `Id`, de `Guid` para `long`.

Resolução 5.8: classe que mapeia a tabela Fornecedores para objetos

```
public class Fornecedor {
    public long Id { get; set; }
    public string Nome { get; set; }
    public string CNPJ { get; set; }
    //Código complementar omitido
}
```

De maneira simplificada, o **Mapeamento Objeto Relacional** (ORM – *Object Relation Mapping*) consiste na existência de uma classe que tenha em suas propriedades os campos existentes em uma tabela, ou o contrário disso: que uma tabela possua campos que possam receber os valores das propriedades de um objeto. Existem diversas ferramentas que automatizam esta atividade, adicionando poderosos recursos. A Microsoft oferece o **Entity Framework** (EF), que é apresentado no capítulo 8.

Nas implementações anteriores, a conexão era obtida de maneira direta, o que não é correto. É importante reforçar que cada classe deve possuir uma única responsabilidade, e que cada método desta classe realize uma única atividade também. A classe onde os códigos implementados estão é uma classe controladora, que recebe e delega interações na interface com o usuário.

No que diz respeito à conexão com o banco, apresento, a seguir, uma nova classe criada.

Resolução 5.9: classe que implementa o padrão de projeto (Design Pattern) Singleton para fornecer uma conexão com o banco de dados

```
public sealed class DBConnection {
    private static volatile SqlConnection instance;

    private DBConnection() { }

    public static SqlConnection DB_Connection {
        get {
            if (instance == null) {
                instance = new SqlConnection(
                    ConfigurationManager.ConnectionStrings[
                    "CS_ADO_NET"].ConnectionString);
            }
            return instance;
        }
    }
}
```

O uso da palavra reservada `sealed` na definição de uma classe garante que ela não seja estendida. Ou seja, nenhuma classe pode tê-la como classe ancestral. O uso na classe se dá ao fato de ela ser específica para ser acessada da maneira que é, sem necessitar de especializações.

Ao se definir um campo da classe ou um método como `static`, define-se que o elemento pertence à classe, e não a um objeto específico. Isso significa que não há necessidade de instanciar a classe para acessar o membro desejado. Desta maneira, todo objeto também tem acesso ao membro, compartilhando seu estado.

A palavra reservada `volatile` na definição de um campo informa que este pode ser modificado por diversas partes da aplicação ao mesmo tempo.

O campo `instance` declarado manterá nele a conexão que será utilizada por toda a aplicação. Ele é privado e estático, ou seja, pertence à classe e apenas métodos estáticos podem acessá-lo. O objetivo é que, na primeira chamada à propriedade `DB_Connection`, este campo receba seu valor e, em novas invocações, o valor seja apenas recuperado. Isso dá maior agilidade à obtenção da conexão.

Pelo fato de a classe implementar o padrão **Singleton**, ela não pode ser instanciada. Desta maneira, seu construtor padrão precisa ser explicitamente

implementado e ser privado à classe, como pode ser verificado na implementação.

`Singleton` é um padrão de projeto que garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto. Na implementação dessa resolução, o ponto global de acesso é a propriedade de leitura `DB_Connection`, que fornecerá sempre a mesma conexão para qualquer chamada a ele. Não faz parte do escopo do livro detalhar padrões de projetos e tampouco afirmar se um padrão é bom ou não. Entretanto, é importante que o leitor busque informações sobre esse tema, pois existem diversos livros que o trabalham.

A propriedade `DB_Connection`, que é apenas de leitura, é pública e estática, pois acessará o campo também estático, `instance`. Inicialmente, verifica-se se o campo é nulo e, caso seja, a instanciação da classe `SqlConnection` ocorrerá, enviando para seu construtor a string de conexão existente no `App.Config`.

Com a implementação da classe de negócio (`Fornecedor`) e da classe para obtenção da conexão com o banco (`DBConnection`), resta agora implementar a busca pelo fornecedor que estiver selecionado no `DataGridView`, o que veremos a seguir.

Resolução 5.10: método que recupera um fornecedor na tabela, com base no Id (Versão 1)

```
private Fornecedor GetFornecedorById(long id) {
    Fornecedor fornecedor = new Fornecedor();
    var connection = DBConnection.DB_Connection;
    var command = new SqlCommand("select id, cnpj, nome
        from FORNECEDORES where id = @id", connection);
    command.Parameters.AddWithValue("@id", id);
    connection.Open();
    using (SqlDataReader reader=command.ExecuteReader()) {
        while (reader.Read()) {
            fornecedor.Id = reader.GetInt32(0);
            fornecedor.CNPJ = reader.GetString(1);
            fornecedor.Nome = reader.GetString(2);
        }
```

```
    }
    connection.Close();
    return fornecedor;
}
```

Para mapear o registro que contém o fornecedor desejado, um objeto da classe `Fornecedor` é criado, e será retornado com os dados do registro obtido.

É importante ressaltar que, pelo código, o método sempre retornará um objeto, pois a pesquisa sempre ocorrerá com base em um registro existente. Para que o método possa ser usado de maneira genérica – na qual, por exemplo, o usuário digite um `Id` e esse valor possa não existir na tabela –, é preciso que o método retorne `null` quando não encontrar um fornecedor na tabela. Procure adaptar o método para esta situação.

Em relação à conexão e execução da instrução SQL, a mudança está na maneira de se obter a conexão, que agora faz uso da classe `DBConnection`. Note que não foi feito uso da instrução `using`, como apresentada anteriormente para a conexão, pois isso causaria a "destruição" do objeto e, pela adoção do `Singleton`, isso não é desejado. Também não foi feito uso do `SqlDataAdapter`, mas sim do `SqlCommand`, que também já foi utilizado, pois a necessidade é de recuperar um dado para ser usado no código, e não popular um controle diretamente.

Após a abertura da conexão, um objeto `SqlDataReader` é usado em uma instrução `using` para receber o resultado retornado pelo método `ExecuteReader()` do `command`, referente à instrução SQL `select`. Com a obtenção do `reader`, é preciso realizar a leitura do conjunto de registros recebidos por ele. Esta é realizada pelo método `Read()`.

Na **Resolução 5.10**, a leitura está presente no argumento da instrução `while()`. No caso do método e da funcionalidade a ser cumprida por ele, não haveria necessidade do `while()`, pois existe a garantia de existência do registro, e apenas um registro será retornado. Na leitura dos dados de cada coluna, é realizada a invocação a um método do tipo do dado a ser lido, enviando como parâmetro o índice da coluna. Por esta necessidade de informar o índice, é importante que as colunas sejam informadas no `select` da instrução SQL. Após a leitura e a atribuição dos valores ao objeto, como se faz

uso de `using`, o `reader` é automaticamente fechado e encerrado. Por fim, a conexão é também fechada e o objeto `fornecedor` retornado.

A classe `SqlDataReader` fornece recursos para leitura de um conjunto de registros de uma tabela SQL Server. O conjunto retornado pode ser lido apenas em um único caminho, do início ao fim (também chamado de *forward-only*). Para a leitura do conjunto existente em um `reader`, é feito uso do método `Read()`, que lê um registro por vez.

A leitura de cada coluna pode ser feita por métodos que representam o tipo de dado da coluna a ser recuperada. Dessa forma, é enviado ao método o valor ordinal dela, ou ainda o nome da coluna com índice de uma matriz, como, por exemplo: **reader[“nomecoluna”]**. Nesta última sintaxe, o retorno é um objeto que representa o tipo nativo da coluna. Após o trabalho com o `reader`, é importante fechá-lo (`Close()`) e eliminá-lo (`Dispose()`), lembrando da facilidade de utilizar o `using`, que executa isso automaticamente ao seu término.

Como o método para obter o fornecedor selecionado no `DataGridView` já está implementado, é necessário chamá-lo. Para isso, o evento `CellClick` foi escolhido e teve seu comportamento implementado, conforme vemos na sequência.

Resolução 5.11: método que captura o evento CellClick do DataGridView (Versão 1)

```
private void dgvFornecedores_CellClick(object sender,
        DataGridViewCellEventArgs e) {
    if (e.RowIndex < 0 || e.ColumnIndex < 0)
        return;

    this.fornecedorAtual = GetFornecedorById(Convert.
        ToInt64(dgvFornecedores.Rows[e.RowIndex].Cells[0].
        Value));
    txtID.Text = this.fornecedorAtual.Id.ToString();
    txtNome.Text = this.fornecedorAtual.Nome;
    txtCNPJ.Text = this.fornecedorAtual.CNPJ;
}
```

O evento CellClick ocorre quando qualquer parte de uma célula é clicada, incluindo bordas e espaços em branco, ou no momento em que o usuário pressiona a tecla de espaço quando uma célula é um botão ou um checkbox que tenha foco; ocorrendo duas vezes para estes tipos de célula se elas forem clicadas enquanto se pressiona a tecla de espaço.

Com a implementação desse método, a cada clique do mouse em qualquer célula do DataGridView, o registro referente ao Id do fornecedor da linha que representa a célula selecionada será buscado. Entretanto, as células que representam o cabeçalho de linhas e colunas também podem ser clicadas e têm este evento capturado. Por esse motivo, caso o disparo do evento tenha ocorrido em uma dessas células, o processamento é interrompido pela invocação da instrução return. Caso a célula seja de conteúdo, o método GetFornecedorById() é invocado, recebendo o Id da linha selecionada. A atribuição dos dados obtidos pelo método aos componentes da tela é uma operação já conhecida.

O objeto fornecedorAtual foi inserido como campo na classe. Veja na sequência.

Resolução 5.12: declarações de membros da classe (Versão 2)

```
public partial class FornecedoresCRUD : Form {
    private string connectionString = ConfigurationManager.
        ConnectionStrings["CS_ADO_NET"].ConnectionString;
    private Fornecedor fornecedorAtual;
    //Código complementar omitido
}
```

Com a implementação da classe DBConnection, para obter a conexão com o banco, o campo connectionString torna-se desnecessário. Desta maneira, adapte os métodos já implementados para que façam uso da nova classe e remova a definição do campo.

5.7 Removendo da tabela o fornecedor selecionado

Com a possibilidade de selecionar um determinado fornecedor no `DataGridView`, agora é possível remover da tabela de fornecedores um registro em específico. Para esta implementação, uma nova classe foi implementada, a `DAL_Fornecedor`, que implementa um *Data Access Layer* (DAL), como pode ser verificado a seguir.

Resolução 5.13: classe DAL_Fornecedor com o método para remoção de um fornecedor em específico

```
public class DAL_Fornecedor {
    private SqlConnection connection = DBConnection.
        DB_Connection;

    public void RemoveById(long id) {
        var command = new SqlCommand("delete from
            FORNECEDORES where id = @id", connection);
        command.Parameters.AddWithValue("@id", id);
        connection.Open();
        command.ExecuteNonQuery();
        connection.Close();
    }
}
```

É possível verificar que, na definição da classe, existe um campo que representa a conexão em que esta trabalhará. O método `RemoveById()` tem como novidade apenas a chamada ao método `ExecuteNonQuery()` do `command`, que executará a instrução SQL `delete`.

Um *Data Access Layer* é uma camada que representa de maneira simplificada o acesso a dados em um armazenamento persistente, como um servidor de banco de dados.

Uma nova alteração na declaração é necessária para utilizar a classe que representa o `DAL`, o que veremos a seguir.

Resolução 5.14: declarações de membros da classe (Versão 3)

```
public partial class FornecedoresCRUD : Form {
    private Fornecedor fornecedorAtual;
    private DAL_Fornecedor dal = new DAL_Fornecedor();
    //Código complementar omitido
}
```

Para efetivamente realizar a remoção do fornecedor selecionado da tabela, é preciso implementar o método que captura o evento `Click` do respectivo botão, como pode ser verificado na sequência.

Resolução 5.15: método que captura o evento Click do botão Remover (Versão 1)

```
private void BtnRemover_Click(object sender, EventArgs e) {
    if (txtID.Text == string.Empty) {
        MessageBox.Show("Selecione o FORNECEDOR a ser
            removido no GRID");
    } else {
        this.dal.RemoveById(this.fornecedorAtual.Id);
        ClearControls();
        MessageBox.Show("Fornecedor removido com sucesso");
    }
}
```

Lendo rapidamente todo o código, é possível notar que, na verificação de existência de um fornecedor selecionado, foi feito uso do controle `txtID` e, no envio do valor do `Id` do fornecedor selecionado para o método `RemoveById()`, foi enviado o valor da propriedade `Id` do objeto que representa o fornecedor atualmente selecionado. Estes dois usos foram apenas para exemplificar; o recomendado é sempre utilizar o objeto e sua propriedade.

Ao concluir a remoção do registro, é preciso que os registros exibidos no `DataGridView` sejam atualizados, pois não pode ser exibido ao usuário um registro já removido.

O método `ClearControls()` sofreu alterações, e sua nova versão pode ser visualizada a seguir.

Resolução 5.16: método ClearControls() (Versão 2)

```
private void ClearControls() {
    txtID.Text = string.Empty;
    txtNome.Text = string.Empty;
    txtCNPJ.Text = string.Empty;
    GetAllFornecedores();
    dgvFornecedores.ClearSelection();
    this.fornecedorAtual = null;
    txtNome.Focus();
}
```

É possível verificar a inserção de três novas instruções, sendo elas: `GetAllFornecedores()`, que deve sair da implementação de inserção; `dgvFornecedores.ClearSelection()`, para garantir que não se ache registro selecionado ao limpar os controles; e `this.fornecedorAtual = null`, que limpa o objeto que representa o fornecedor atualmente selecionado.

5.8 Realizando alterações em um fornecedor já inserido

A última operação do CRUD a ser implementado é o de atualização de dados. Esta atualização refere-se a mudanças em dados que já foram inseridos na tabela. Seguindo o padrão do capítulo 4, ela ocorrerá por meio do método do botão `Gravar`, o mesmo usado para inserção. O método responsável pela atualização dos dados pode ser visto na sequência.

Resolução 5.17: método para atualização de dados na classe DAL_Fornecedor

```
private void Update(Fornecedor fornecedor) {
    var command = new SqlCommand("update FORNECEDORES set
        cnpj=@cnpj, nome=@nome where id=@id",
        this.connection);
    command.Parameters.AddWithValue("@cnpj", fornecedor.
        CNPJ);
```

```
        command.Parameters.AddWithValue("@nome", fornecedor.
            Nome);
        command.Parameters.AddWithValue("@id", fornecedor.Id);
        connection.Open();
        command.ExecuteNonQuery();
        connection.Close();
}
```

O método `Update()` da classe `DAL_Fornecedor` recebe o objeto que tem os dados atualizados e que precisam ser persistidos na tabela. As instruções do método são todas já conhecidas. Entretanto, é possível verificar na definição do método que ele está como privado (`private`), ou seja, acessível apenas por métodos da própria classe. A seguir, apresento o método chamador.

Resolução 5.18: método que delega a atualização ou inserção de registro na classe DAL_Fornecedor

```
public void Save(Fornecedor fornecedor) {
    if (fornecedor.Id != null)
        this.Update(fornecedor);
    else {
        this.Insert(fornecedor);
    }
}
```

A classe `DAL` oferecerá um único método para gravar o fornecedor na tabela, seja a inserção de um novo fornecedor ou a atualização de dados de um já existente. Para identificar qual operação será realizada, foi adotada a propriedade `Id` como *flag*. Se essa propriedade não for nula, é porque o dado já foi inserido, uma vez que o campo que mapeia essa propriedade é autoincremento.

Recorde-se que o campo `Id`, na classe `Fornecedor`, é definido como `long` e, este tipo de dado é visto como primitivo. Sendo assim, possui um valor *default*, que é zero, não aceitando, assim, um valor nulo. Entretanto, o C# oferece um recurso que permite que isso ocorra. Este é exibido na sequência, que apresenta uma nova versão para a classe `Fornecedor`.

Resolução 5.19: classe Fornecedor com propriedade Id aceitando valor nulo

```
public class Fornecedor {
    public long? Id { get; set; }
    public string Nome { get; set; }
    public string CNPJ { get; set; }

    public Fornecedor() {
        this.Id = null;
    }
}
```

Veja na definição da propriedade `Id` que, após o nome do tipo de dado a que ela se refere, é informado o símbolo de interrogação. Com esta sintaxe, a propriedade passa a aceitar valores nulos. Também foi criado um construtor padrão de maneira explícita, para que, ao inicializar um objeto, o valor nulo seja atribuído à propriedade.

Mudanças também são necessárias no método de gravação do fornecedor, pois, na versão atual, a inserção é realizada nele, e agora foi transferida para o DAL, o que veremos a seguir.

Resolução 5.20: método que implementa o evento Click do botão Gravar (Versão 3)

```
private void btnGravar_Click(object sender, EventArgs e) {
    dal.Save(new Fornecedor() {
        Id = string.IsNullOrEmpty(txtID.Text) ?
            (long?) null : Convert.ToInt64(txtID.Text),
        Nome = txtNome.Text,
        CNPJ = txtCNPJ.Text
    });
    MessageBox.Show("Manutenção realizada com sucesso");
    ClearControls();
}
```

Verifique que o método agora tem a responsabilidade de instanciar um objeto da classe `Fornecedor` e enviá-lo ao método `Save()` do `DAL()`.

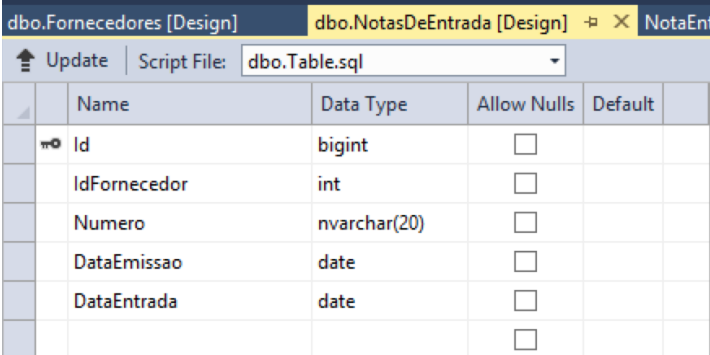
Note que na inicialização da propriedade `Id` é feito uso do operador ternário condicional, onde é enviado `null` caso não seja uma alteração. Para efeito de sintaxe, o `null` é convertido para o tipo de dado da propriedade.

Com a mudança do tipo da propriedade `Id` para `long?`, o método da classe `DAL RemoveById()` precisará de atualização no tipo de dado que recebe, para que seja o mesmo da propriedade, ou seja, `long?`. Faça isso.

Uma vez alterado o método do botão `Gravar`, a inserção precisa ser transferida para o `DAL`, da mesma maneira que existe o método `Update()`. Faça isso, lembrando que o método precisa ser `private`.

Para finalizar a implementação da janela responsável pela interação e manutenção com dados de fornecedores, codifique o método para o evento `Click` dos botões `Novo` e `Cancelar`, da mesma maneira que foi feito no capítulo 4 (**Resoluções 4.20** e **4.24** da seção 4.7, respectivamente).

5.9 Implementando associações em tabelas de uma base de dados

Seguindo ainda o exemplo do capítulo 4, é preciso agora trazer para a solução a classe que representa a nota de entrada de uma compra, nomeada `NotaEntrada`, apresentada na sequência.

Resolução 5.21: classe NotaEntrada (Versão 1)

```
public class NotaEntrada {
    public long? Id { get; set; }
    public string Numero { get; set; }
    public Fornecedor FornecedorNota { get; set; }
    public DateTime DataEmissao { get; set; }
    public DateTime DataEntrada { get; set; }

    public NotaEntrada() {
        this.Id = null;
    }
    // Código não implementado
}
```

Comparando a classe dessa resolução com a final, implementada no capítulo 4 (**Resolução 4.8** da seção 4.5), verifica-se a mudança no tipo de dado do `Id`, a retirada (no momento) do atributo `Produtos` e a definição do construtor padrão de maneira explícita.

Com a classe que representa a tabela criada, é preciso agora criar a tabela. Siga os passos apresentados para a criação da tabela `Fornecedores` para criar a tabela `NotasDeEntrada`, com a estrutura igual à apresentada pela figura:

dbo.Fornecedores [Design] | dbo.NotasDeEntrada [Design] | NotaEnt

Update | Script File: dbo.Table.sql

Name	Data Type	Allow Nulls	Default
Id	bigint	☐	
IdFornecedor	int	☐	
Numero	nvarchar(20)	☐	
DataEmissao	date	☐	
DataEntrada	date	☐	
		☐	

Fig. 5.16: Estrutura criada e salva para a tabela NotasDeEntrada

Verifique no nome da tabela que ela já está criada na base de dados já atualizada. Para atualizar a base de dados para essa criação, clique no botão `Update`, logo acima da estrutura. Note o campo `IdFornecedor`, pois é por meio dele que a associação da classe `Fornecedor` com `NotaEntrada` será persistida como forma de **relacionamento**.

Quando se modela uma base de dados, define-se um diagrama chamado **DER** (Diagrama de Entidades e Relacionamentos), no qual se busca relacionar as tabelas – aqui nomeadas entidades – que possuem algum tipo de ligação.

Quando se trabalha com outra ferramenta visual, o **MER** (Modelo de Entidades e Relacionamentos), define-se uma coluna que servirá como chave estrangeira (*Foreign key*), nas tabelas que serão dependentes no relacionamento documentado. Este campo tem, inicialmente, a finalidade de apontar para o

registro da tabela – ou "relação", neste contexto – primária, dona do relacionamento, o registro que é responsável pelo dado armazenado na coluna de chave estrangeira.

Desta maneira, estabelece-se uma regra, onde o dado da coluna de chave estrangeira deve existir obrigatoriamente na coluna de chave primária da tabela que origina a chave estrangeira, ou seja, cria-se uma `constraint`. Este assunto não se esgota aqui, pois é vasto e merece um estudo detalhado, o que está fora do contexto deste livro.

Uma associação, para ser mapeada para uma tabela em uma base de dados, precisa de um relacionamento entre as tabelas envolvidas. Para criar o relacionamento entre as tabelas `Fornecedores` e `NotasDeEntrada` nos itens exibidos ao lado da estrutura definida, clique com o botão direito sobre `Foreign Keys` e, então, em `Add New Foreign Key`:

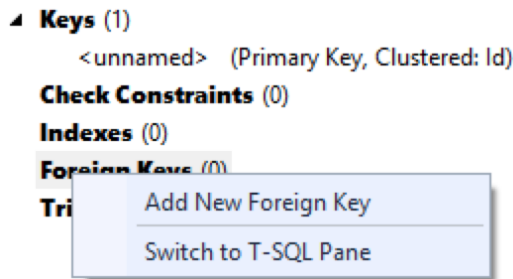



Fig. 5.17: Adicionando uma nova chave estrangeira

Após confirmar a opção do menu de contexto, uma nova chave estrangeira é criada, com o nome dela em edição; altere-o para `FK_NotasDeEntrada_ToFornecedor`, como vemos:

Keys (1)
<unnamed> (Primary Key, Clustered: Id)
Check Constraints (0)
Indexes (0)
Foreign Keys (1)
FK_NotasDeEntrada_ToFornecedor (ToTableColumn)
Triggers (0)

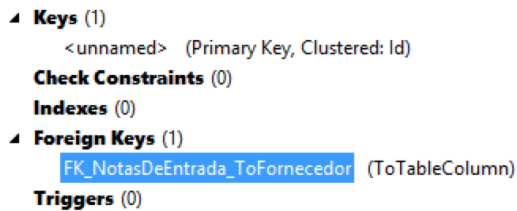



Fig. 5.18: Nomeando a chave estrangeira criada

Tendo criado e nomeado a chave estrangeira, é preciso configurá-la. Para isso, será preciso alterar o código na guia `T-SQL`, pois o SSDT não oferece ferramenta visual para essa funcionalidade, veja a figura a seguir:

Design ↑↓ T-SQL

```
CREATE TABLE [dbo].[NotasDeEntrada]
(
    [Id] BIGINT NOT NULL PRIMARY KEY,
    [IdFornecedor] BIGINT NOT NULL,
    [Numero] NVARCHAR(20) NOT NULL,
    [DataEmissao] DATE NOT NULL,
    [DataEntrada] DATE NOT NULL,
    CONSTRAINT [FK_NotasDeEntrada_ToFornecedor]
        FOREIGN KEY ([IdFornecedor]) REFERENCES [Fornecedores]([Id])
)
```

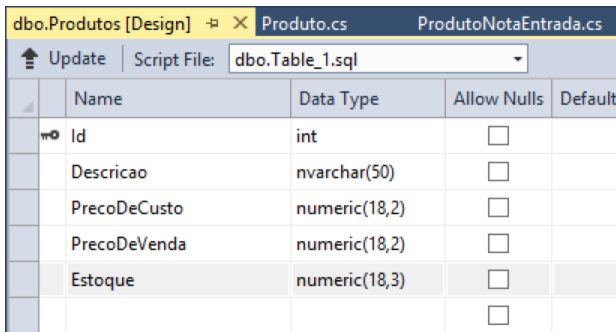



Fig. 5.19: Alterando o código SQL para configuração da chave estrangeira

A configuração da *foreign key* pode ser vista na última linha de código da figura 5.19.

Crie uma classe `Produto` semelhante à classe de mesmo nome criada no capítulo 4 (**Resolução 4.5** da seção 4.3). Lembre-se de mudar o tipo de dado da propriedade `Id` e da criação do construtor que inicializará essa propriedade, da mesma maneira que foi feito para as duas classes criadas anteriormente.

Com a classe `Produto` criada, a classe que se associará à `NotaEntrada` também pode ser criada. Implemente a classe `ProdutoNotaEntrada`, seguindo as orientações descritas para a criação de `Produto`.

É preciso agora criar a tabela que será mapeada para a classe `Produto`. Da mesma maneira que foi criada a tabela `Fornecedores` anteriormente, crie agora a tabela `Produtos`, que tem sua estrutura apresentada pela figura:

dbo.Produtos [Design] | Produto.cs | ProdutoNotaEntrada.cs

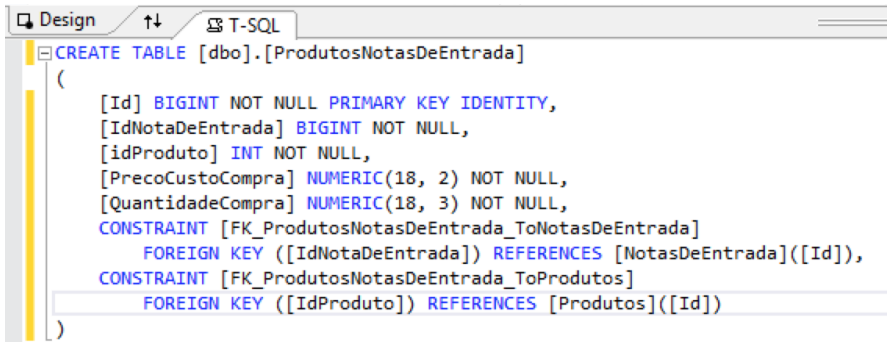
Update | Script File: dbo.Table_1.sql

Name	Data Type	Allow Nulls	Default
Id	int	☐	
Descricao	nvarchar(50)	☐	
PrecoDeCusto	numeric(18,2)	☐	
PrecoDeVenda	numeric(18,2)	☐	
Estoque	numeric(18,3)	☐	
		☐	

Fig. 5.20: Estrutura necessária para a tabela Produtos

Com a tabela e a classe que se referenciam a `Produtos` implementadas e criadas, é preciso agora criar a interface com o usuário para que ele possa realizar manutenções e inserções de dados na tabela. Seguindo o exemplo visual apresentado no capítulo 4, nesta nova solução, implemente a nova janela, já interagindo com a base de dados. Aproveite e crie também a janela principal para a aplicação e os menus de acesso às janelas.

Da mesma maneira como foi feito com as tabelas `Fornecedores` e `NotasDeEntrada`, é preciso agora relacionar a tabela que representará os produtos recebidos em cada nota de entrada. Para isso, crie a tabela `ProdutosNotasDeEntrada` de acordo com o apresentado na figura 5.21, que se refere ao código completo, necessário para a criação.

```
Design  ↑↓  T-SQL
CREATE TABLE [dbo].[ProdutosNotasDeEntrada]
(
    [Id] BIGINT NOT NULL PRIMARY KEY IDENTITY,
    [IdNotaDeEntrada] BIGINT NOT NULL,
    [idProduto] INT NOT NULL,
    [PrecoCustoCompra] NUMERIC(18, 2) NOT NULL,
    [QuantidadeCompra] NUMERIC(18, 3) NOT NULL,
    CONSTRAINT [FK_ProdutosNotasDeEntrada_ToNotasDeEntrada]
        FOREIGN KEY ([IdNotaDeEntrada]) REFERENCES [NotasDeEntrada]([Id]),
    CONSTRAINT [FK_ProdutosNotasDeEntrada_ToProdutos]
        FOREIGN KEY ([IdProduto]) REFERENCES [Produtos]([Id])
)
```

Fig. 5.21: Estrutura necessária para a tabela ProdutosNotasDeEntrada

Note nos campos da tabela a existência do campo `IdNotaDeEntrada`, que não tem na classe que mapeia essa tabela uma propriedade semelhante. Ao final da instrução `CREATE TABLE`, identificam-se as duas chaves estrangeiras necessárias para implementar a associação com a tabela `Produtos` e para que uma `NotaDeEntrada` saiba quais são seus `ProdutosDeNotasDeEntrada`.

Uma vez criadas as tabelas na base de dados, pode ocorrer a necessidade de realizar alguma alteração em sua estrutura, ou, ainda, a necessidade de inserir registros de maneira manual nelas. Para realizar qualquer uma dessas operações, basta, no `Server Explorer`, clicar com o botão direito do mouse sobre o nome da tabela e escolher a opção desejada: `Show Table Definition` para exibir a estrutura da tabela e, então, realizar modificações; ou `Show Table Data` para ver os registros inseridos, alterá-los ou inserir novos:

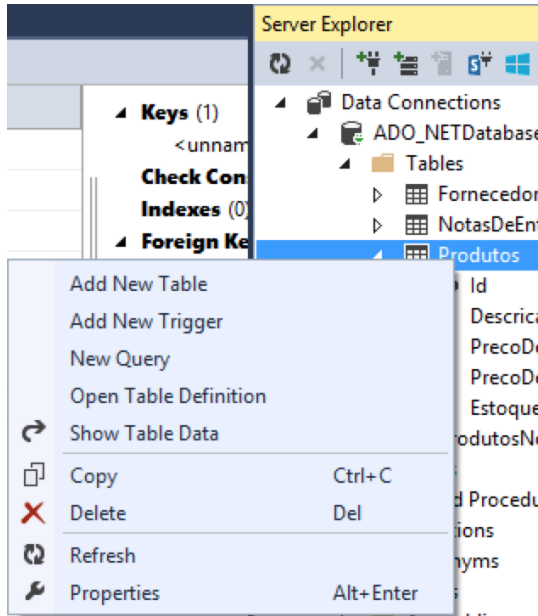



Fig. 5.22: Visualizando operações possíveis de realizar em uma tabela já criada

Agora com as classes e tabelas criadas, implemente na classe `NotaEntrada` a propriedade `Produtos`, sua inicialização no construtor e os métodos `RegistrarProduto()`, `RemoverProduto()` e `RemoverTodosProdutos()`, da mesma maneira que eles foram implementados no capítulo 4 na **Resolução 4.8** (seção 4.5).

Com todas as classes implementadas, é preciso criar a interface com o usuário para o registro da nota de entrada. Implemente o formulário de acordo com o já criado no capítulo 4 (figura 4.8 da seção 4.8).

A implementação dos métodos que serão responsáveis pela implementação do relacionamento entre as tabelas que mapeiam as associações entre as classes `Produto`, `Fornecedor`, `NotaEntrada` e `ProdutoNotaEntrada` será realizada na classe `DAL_NotaEntrada`, que tem sua declaração e seu primeiro método apresentado na sequência.

Resolução 5.22: método para inserção de uma nota de entrada na tabela

```
public class DAL_NotaEntrada {
    private SqlConnection connection = DBConnection.
        DB_Connection;

    private void Insert(NotaEntrada notaEntrada) {
        var command = new SqlCommand("insert into " +
            "NOTASDEENTRADA(IdFornecedor, Numero, "+
            "DataEmissao, "DataEntrada) values(" +
            "@IdFornecedor, @Numero, @DataEmissao, " +
            "@DataEntrada)", connection);
        command.Parameters.AddWithValue("@IdFornecedor",
            notaEntrada.FornecedorNota.Id);
        command.Parameters.AddWithValue("@Numero",
            notaEntrada.Numero);
        command.Parameters.AddWithValue("@DataEmissao",
            notaEntrada.DataEmissao);
        command.Parameters.AddWithValue("@DataEntrada",
            notaEntrada.DataEntrada);
        connection.Open();
        command.ExecuteNonQuery();
        connection.Close();
    }
}
```

Note que, na inserção da nota de entrada informada pelo usuário, não existe referência aos produtos da nota. Isso deve-se à regra de que, para inserção de produtos em uma nota, ela já deve existir na tabela, tal como definido no capítulo 4. O segundo método a ser implementado na classe `DAL` será o de `Update()`, que é apresentado a seguir.

Resolução 5.23: método que atualiza dados de uma determinada nota de entrada na tabela

```
private void Update(NotaEntrada notaEntrada) {
    var command = new SqlCommand("update NOTASDEENTRADA " +
        "set IdFornecedor=@IdFornecedor, Numero=@Numero, "+
```

```
        "DataEmissao=@DataEmissao, DataEntrada=" +
        "@DataEntrada where (Id=@Id)", connection);
    command.Parameters.AddWithValue("@IdFornecedor",
        notaEntrada.FornecedorNota.Id);
    command.Parameters.AddWithValue("@Numero",
        notaEntrada.Numero);
    command.Parameters.AddWithValue("@DataEmissao",
        notaEntrada.DataEmissao);
    command.Parameters.AddWithValue("@DataEntrada",
        notaEntrada.DataEntrada);
    command.Parameters.AddWithValue("@Id",
        notaEntrada.Id);
    connection.Open();
    command.ExecuteNonQuery();
    connection.Close();
    DeleteAllProdutosFromNotaEntrada(notaEntrada.Id);
    InsertProdutosNotaDeEntrada(notaEntrada.Id,
        notaEntrada.Produtos);
}
```

A penúltima instrução do método `Update()` é a invocação ao método `DeleteAllProdutosFromNotaEntrada()`, que recebe como argumento o `Id` da nota de entrada. Este método será responsável por remover da tabela `ProdutosNotasDeEntrada` todos os registros que pertencem a nota que está sendo atualizada (**Resolução 5.24**).

A última instrução invoca o método `InsertProdutosNotaDeEntrada()`, que recebe o `Id` da nota e a coleção atual de `Produtos` que esta possui (**Resolução 5.25**). Este método será responsável por inserir na tabela `ProdutosNotasDeEntrada` a coleção atualizada de produtos.

Resolução 5.24: método que remove da tabela todos os produtos de uma determinada nota de entrada

```
private void DeleteAllProdutosFromNotaEntrada(long?
        idNotaEntrada) {
    var command = new SqlCommand("delete from " +
        NOTASDEENTRADA where (Id=@Id)", connection);
```

```
    command.Parameters.AddWithValue("@Id",
        idNotaEntrada);
    connection.Open();
    command.ExecuteNonQuery();
    connection.Close();
}
```

Resolução 5.25: método que insere da tabela todos os produtos de uma nota de entrada

```
private void InsertProdutosNotaDeEntrada(long? idNotaEntrada,
        IList<ProdutoNotaEntrada> produtos) {
    var command = new SqlCommand("insert into " +
        "PRODUTOSNOTASDEENTRADA(IdNotaDeEntrada, " +
        "IdProduto, PrecoCustoCompra, QuantidadeCompra) " +
        "values(@IdNotaDeEntrada, @IdProduto, " +
        "@PrecoCustoCompra, @QuantidadeCompra",
        connection);
    connection.Open();
    foreach (var produto in produtos) {
        command.Parameters.Clear();
        command.Parameters.AddWithValue("@IdNotaDeEntrada",
            idNotaEntrada);
        command.Parameters.AddWithValue("@IdProduto",
            produto.Id);
        command.Parameters.AddWithValue(
            "@PrecoCustoCompra", produto.PrecoCustoCompra);
        command.Parameters.AddWithValue(
            "@QuantidadeCompra",produto.
            QuantidadeComprada);
        command.ExecuteNonQuery();
    }
    connection.Close();
}
```

Verifique o uso da instrução `foreach`, que percorre a coleção recebida como argumento. A cada produto existente na coleção, os parâmetros são limpos, inseridos de acordo com cada produto, por meio da chamada ao método `ExecuteNonQuery()`.

O próximo método a ser implementado é o que será invocado pela interface com o usuário, o `Save()`, que delegará a execução para o método `Update()` ou `Insert()`, de acordo com o objeto recebido como argumento. Este pode ser visualizado na sequência.

Resolução 5.26: método responsável por delegar a atualização de dados de uma nota de Entrada para inserção ou atualização de dados

```
public void Save(NotaEntrada notaEntrada) {
    if (notaEntrada.Id == null)
        this.Insert(notaEntrada);
    else
        this.Update(notaEntrada);
}
```

5.10 Implementando a interface com o usuário para as associações

Com as tabelas criadas e os principais métodos para a classe `DAL_NotaEntrada` já implementados, é preciso implementar a janela da Nota de Entrada. O primeiro passo é a definição dos `DAL`s que serão utilizados na janela, o que veremos a seguir.

Resolução 5.27: declaração de campos para a classe

```
public partial class NotasEntradaCRUD : Form {
    private DAL_NotaEntrada dal = new DAL_NotaEntrada();
    private DAL_Fornecedor dalFornecedor =
        new DAL_Fornecedor();
    private DAL_Produto dalProduto = new DAL_Produto();
    private NotaEntrada notaAtual;
    // Código omitido
}
```

A janela de Nota de Entrada possui dados vindos de duas tabelas: `Fornecedores` e `Produtos`. Para selecionar o fornecedor da nota e o pro-

duto da compra, será feito uso de `ComboBox`, da mesma maneira que foi feito no capítulo 4, com a diferença de que agora os dados vêm de tabelas, e não de coleções. O preenchimento desses `ComboBoxs` é realizado, inicialmente, no construtor da classe, visto na sequência.

Resolução 5.28: método construtor da interface com o usuário para o registro das notas de entrada

```
public NotasEntradaCRUD() {
    InitializeComponent();
    InicializaComboBoxs();
}
```

Pelo fato de a população dos `ComboBoxs` poder ser necessária em diversos momentos da interação com o usuário, foi implementado um método (o `InicializaComboBox()`) para esta funcionalidade, que pode ser visualizado na sequência.

Resolução 5.29: método que inicializa os ComboBoxs

```
private void InicializaComboBoxs() {
    cbxFornecedor.Items.Clear();
    cbxProduto.Items.Clear();
    foreach (Fornecedor fornecedor in this.dalFornecedor.
            GetAllAsIList()) {
        cbxFornecedor.Items.Add(fornecedor);
    }

    foreach (Produto produto in this.dalProduto.
            GetAllAsIList()) {
        cbxProduto.Items.Add(produto);
    }
}
```

Observe no código que a primeira funcionalidade do método é realizar a limpeza dos itens de cada `ComboBox`. Tanto essa limpeza como a chamada periódica a esse método deve-se ao fato de os dados virem de uma tabela e de

que, um outro usuário, acessando a aplicação ao mesmo tempo, pode inserir novos dados e, na sua execução, eles podem ser necessários.

Um segundo ponto importante é a coleção de dados vinda dos `DALs`, representada por métodos que retornam todos os dados, como uma coleção do tipo `IList`. A implementação do método `GetAllAsIList()` da classe `DAL_Fornecedor` pode ser visualizada a seguir.

Resolução 5.30: método que recupera todos os fornecedores da tabela de dados e retorna como um objeto IList

```
public IList<Fornecedor> GetAllAsIList() {
        IList<Fornecedor> fornecedores =
    new List<Fornecedor>();

    var adapter = new SqlDataAdapter("select id, cnpj,
        nome from FORNECEDORES", connection);
    var builder = new SqlCommandBuilder(adapter);

    var table = new DataTable();
    adapter.Fill(table);
    connection.Close();

    for (int i = 0; i < table.Rows.Count; i++) {
        var row = table.Rows[i];
        fornecedores.Add(
            new Fornecedor() {
                Id = Convert.ToInt64(row["id"]),
                CNPJ = (string) row["cnpj"],
                Nome = (string) row["nome"]
        });
    }
    return fornecedores;
}
```

Observe no bloco de repetição `for()` que é feito uso da quantidade (`Count`) de linhas (`Rows`) populadas no `DataTable`, como término da interação. As instruções anteriores são todas já conhecidas e dispensam explicações. Note que a navegação entre os registros é realizada após o fechamento

da conexão com o banco, pois os dados já estão na memória. Esta é uma característica do ADO.NET: trabalhar de maneira desconectada.

As instruções que compõem o bloco `for()` começam por obter uma determinada linha (registro da tabela) de acordo com o seu índice (variável de interação `i`). Com a linha obtida, um objeto `Fornecedor` é instanciado e, então, adicionado à coleção de fornecedores. Note que cada campo é acessado na linha por meio de seu nome, identificado como índice (embora seja utilizado string). O retorno de cada coluna é um `object`, por isso a necessidade de *casts*.

Renomeie os métodos `GetAll()` das classes `DAL` para `GetAllAsDataTable()`, pois fica mais condizente com o novo método criado. Desta maneira, é possível ter vários métodos com retornos diferentes e nomes semanticamente compreensíveis.

Crie nas classes `DAL_Produto` e `DAL_NotaEntrada` os métodos `GetAllAsIList()` e `GetAllAsDataTable()`, seguindo os moldes dos métodos de mesmo nome da classe `DAL_Fornecedor`.

Com os `ComboBox`s populados, já é possível realizar a inclusão de notas. Implemente o método que implementa o evento `Click` do botão `Gravar`, de acordo com a resolução que segue.

Resolução 5.31: método que implementa o evento Click do botão Gravar da Nota de Entrada

```
private void btnGravarNota_Click(object sender,
        System.EventArgs e) {
    dal.Save(new NotaEntrada() {
        Id = string.IsNullOrEmpty(txtIDNotaEntrada.
            Text) ? (long?)null : Convert.ToInt64(
            txtIDNotaEntrada.Text),
        Numero = txtNumero.Text,
        DataEmissao = Convert.ToDateTime(dtpEmissao.Value),
        DataEntrada = Convert.ToDateTime(dtpEntrada.Value),
        FornecedorNota = (Fornecedor) cbxFornecedor.
            SelectedItem
    });
    MessageBox.Show("Manutenção realizada com sucesso");
```

```
    ClearControls();
}
```

A lógica do método não é diferente das anteriores. Ela foi exibida apenas para apresentar – embora tenha sido apresentado no capítulo 4 – como obter os dados dos `ComboBox`s.

O método `ClearControls()` segue a mesma lógica dos implementados neste capítulo. Entretanto, os `ComboBox`s e os `DateTimePicker`s têm seus padrões de inicialização. Sendo assim, recorra ao capítulo 4 para ver essas particularidades e implemente esse método (**Resolução 4.33** da seção 4.10).

Para registrar na nota de compra os produtos comprados, é preciso que a nota esteja devidamente persistida na tabela e selecionada no formulário com os dados do corpo da nota. Desta maneira, é possível informar os dados dos produtos comprados. Entretanto, ao iniciar o formulário de entrada, é preciso que os registros já realizados sejam exibidos. Para isso, invoque o método `GetAllNotas()`, exibido na sequência, no construtor da classe que representa a janela de interação com o usuário.

Resolução 5.32: método que recupera todas as notas registradas para exibir no DataGridView

```
private void GetAllNotas() {
    dgvNotasEntrada.DataSource = dal.GetAllAsDataTable();
}
```

A resolução seguinte apresenta o método referente à seleção de uma nota no `DataGridView`.

Resolução 5.33: método que captura o evento CellClick do DataGridView de notas de entrada, para seleção de uma nota

```
private void dgvNotasEntrada_CellClick(object sender,
        DataGridViewCellEventArgs e) {
    if (e.RowIndex < 0 || e.ColumnIndex < 0)
        return;
    this.notaAtual = dal.GetById(Convert.ToInt64(
```

```
        dgvNotasEntrada.Rows[e.RowIndex].Cells[0].Value));
    txtIDNotaEntrada.Text = notaAtual.Id.ToString();
    txtNumero.Text = notaAtual.Numero;
    dtpEmissao.Value = notaAtual.DataEmissao;
    dtpEntrada.Value = notaAtual.DataEntrada;
    cbxFornecedor.SelectedItem = notaAtual.FornecedorNota;
}
```

O método `GetById()`, que realiza a busca da nota de entrada selecionada na tabela, pertence ao `DAL`, diferentemente do apresentado na implementação do formulário de fornecedores. Sua implementação pode ser visualizada a seguir.

Resolução 5.34: método que realiza a busca por uma nota de entrada tendo como base seu Id

```
public NotaEntrada GetById(long id) {
    NotaEntrada notaEntrada = new NotaEntrada();
    DAL_Fornecedor dalFornecedor = new DAL_Fornecedor();
    long idFornecedorNota = -1;
    var command = new SqlCommand("select id, idfornecedor,
        numero, dataemissao, dataentrada from NOTASDEENTRADA
        where id = @id", connection);
    command.Parameters.AddWithValue("@id", id);
    connection.Open();
    using (SqlDataReader reader = command.ExecuteReader()) {
        while (reader.Read()) {
            notaEntrada.Id = reader.GetInt64(0);
            idFornecedorNota = reader.GetInt32(1);
            notaEntrada.Numero = reader.GetString(2);
            notaEntrada.DataEmissao = reader.GetDateTime(3);
            notaEntrada.DataEntrada = reader.GetDateTime(4);
        }
    }
    connection.Close();
    if (idFornecedorNota > 0)
        notaEntrada.FornecedorNota = dalFornecedor.
            GetById(idFornecedorNota);
    return notaEntrada;
```

```
}
```

Ao analisarmos o código, identificamos uma variável, a `idFornecedorNota`. Sua finalidade refere-se ao fato de que, no momento da leitura do `SqlDataReader` que recupera a nota desejada, há a invocação ao método `GetById` da classe `DAL_Fornecedor()`. Os dois métodos compartilham a mesma conexão e, de acordo com as regras para o uso de um `DataReader`, não é possível manter dois objetos dessa classe abertos ao mesmo tempo, para uma mesma conexão. Sendo assim, o `Id` do fornecedor da nota localizada é armazenado em uma variável e, ao término da cláusula `using`, esse valor, caso um fornecedor tenha sido encontrado, é enviado como argumento para o método do `DAL` de fornecedores, o que veremos na sequência.

Resolução 5.35: método que realiza a busca por um fornecedor tendo como base seu Id

```
public Fornecedor GetById(long id) {
    Fornecedor fornecedor = new Fornecedor();
    var command = new SqlCommand("select id, cnpj, nome from
        FORNECEDORES where id = @id", connection);
    command.Parameters.AddWithValue("@id", id);
    connection.Open();

    using (SqlDataReader reader = command.ExecuteReader()) {
        while (reader.Read()) {
            fornecedor.Id = reader.GetInt32(0);
            fornecedor.CNPJ = reader.GetString(1);
            fornecedor.Nome = reader.GetString(2);
        }
    }
    connection.Close();
    return fornecedor;
}
```

Com a possibilidade de registrar os produtos comprados para a nota em edição, é preciso agora implementar esta funcionalidade. Trata-se do evento `Click` do botão `Novo` dos dados de produtos.

Implemente o método que captura o evento `Click` do botão `Novo` referente aos dados do produto a ser informado. Ao implementá-lo, você verá (e recordará) que dois novos métodos também devem ser implementados: o `ChangeStatusOfControls()` e o `ClearControlsProduto()`. Esses três métodos são os mesmos implementados no capítulo 4.

Embora a associação e relacionamento entre classes/tabelas de Notas de Entrada e seus Produtos já tenham sido implementadas, por meio dos métodos relacionados à atualização da nota (`Update()` de `DAL_NotaEntrada`), é possível fazê-lo de uma maneira mais simples, uma vez que cada objeto da classe `ProdutoNotaEntrada` possui seu `Id` único, o que também ocorre na tabela. A seguir, trago o método responsável por inserir o novo produto da compra.

Resolução 5.36: método da classe DAL que realiza a inserção do produto registrado em uma compra

```
private void InsertProduto(NotaEntrada notaEntrada,
        ProdutoNotaEntrada produto) {
    notaEntrada.Produtos.Add(produto);
    var command = new SqlCommand("insert into " +
        "PRODUTOSNOTASDEENTRADA(idnotadeentrada, " +
        "idproduto,  precocustocompra, quantidadecompra) " +
        "values(@idnotadeentrada, @idproduto, " +
        "@precocustocompra, @quantidadecompra)",
        this.connection);
    command.Parameters.AddWithValue("@idnotadeentrada",
        notaEntrada.Id);
    command.Parameters.AddWithValue("@idproduto",
        produto.Id);
    command.Parameters.AddWithValue("@precocustocompra",
        produto.PrecoCustoCompra);
    command.Parameters.AddWithValue("@quantidadecompra",
        produto.QuantidadeComprada);
    connection.Open();
    command.ExecuteNonQuery();
    connection.Close();
}
```

Veja que o método recebe, além do produto que será registrado, a nota a que este se refere. Na classe `ProdutoNotaEntrada` não existe referência para que nota cada produto pertence, mas na tabela existe um campo com o `Id` da nota de entrada. Para que o relacionamento entre essas duas possa ser implementado, é preciso que este método saiba qual é a nota que tem o produto. O método para atualização dos dados de um produto já inserido pode ser verificado na sequência.

Resolução 5.37: método da classe DAL que realiza a atualização na tabela de dados de um produto previamente inserido

```
private void UpdateProduto(ProdutoNotaEntrada produto) {
    var command = new SqlCommand("update " +
        "PRODUTOSNOTASDEENTRADA set idproduto=@idproduto, "+
        "precocustocompra=@precocustocompra, " +
        "quantidadecompra=@quantidadecompra) where " +
        (id=@id)", this.connection);
    command.Parameters.AddWithValue("@idproduto",
        produto.Id);
    command.Parameters.AddWithValue("@precocustocompra",
        produto.PrecoCustoCompra);
    command.Parameters.AddWithValue("@quantidadecompra",
        produto.QuantidadeComprada);
    command.Parameters.AddWithValue("@id", produto.Id);
    connection.Open();
    command.ExecuteNonQuery();
    connection.Close();
}
```

Com os métodos de inserção e atualização implementados, é preciso implementar o método público, que será invocado pelo método que capturar o evento `Click` do botão `Gravar` dos produtos.

Uma vez que o comportamento esperado para o método que captura o evento `Click` do botão `Gravar` dos dados de produtos é o mesmo que o implementado para a nota de entrada – alterando apenas os componentes de origem dos dados e o objeto a ser enviado para o DAL –, implemente-o. Ao final, também precisamos invocar o `ClearControlsProduto()`, e não o `ClearControls()`.

Implemente o comportamento para o método que captura o evento `Click` do botão `Remover` de produtos. Será preciso também implementar o método na classe `DAL`, que será invocado por esse método.

Realize a implementação do método para o evento `Click` do botão `Remover` para remover uma nota. Utilize os métodos já implementados para remover a nota selecionada e os produtos registrados para ela.

5.11 Conclusão

Neste capítulo, foi trabalhado o acesso a bancos de dados. Uma adaptação da solução implementada no capítulo 4 foi proposta. Novos conceitos, recursos da linguagem, ferramentas e técnicas foram apresentados, discutidos e trabalhados.

Agora, com o conhecimento de como manipular uma base de dados, é possível a criação de uma aplicação maior, pois o domínio de conhecimento oferecido até este capítulo traz subsídios para isso. O próximo capítulo apresenta uma nova técnica para acesso a dados, que é o uso de `DataSet` Tipado.

Capítulo 6

Utilizando DataSet Tipado para acesso à base de dados

Alguns dos recursos oferecidos pelo ADO.NET foram apresentados no capítulo 5, onde foram utilizadas as classes `SqlConnection`, `SqlDataAdapter`, `SqlCommand`, `SqlDataReader` e, de maneira bem simples, a `DataTable`, para realizar seleções e manutenções em registros de uma base de dados. Todos os exemplos fizeram uso, de forma explícita, de instruções SQL.

Outro recurso para a manipulação de dados oferecido pelo ADO.NET é o **DataSet**. Inicialmente, serão utilizados exemplos diretos, sem uso de interface com o usuário, para conhecimento e aplicação desse recurso. Ao final, será construída, com aprimoramentos, a aplicação trabalhada no capítulo 5, fazendo uso de DataSet Tipado.

6.1 Introdução

O `DataSet` do ADO.NET é um componente que contém, de forma direta, outros dois componentes: `DataTableCollection` e sua `DataRelationCollection`. Ele representa um conjunto completo de dados, incluindo as tabelas (estruturas) que contêm esses dados, seus relacionamentos, suas classificações e também as restrições que estes possam conter ou sofrer.

É possível afirmar que um `DataSet` representa uma completa base de dados, com tabelas, colunas, registros, `constraints` (restrições), chaves primárias, chaves estrangeiras e índices. Tudo isso localmente, na memória RAM de seu computador. A figura 6.1 apresenta um diagrama, que traz o `DataSet`.

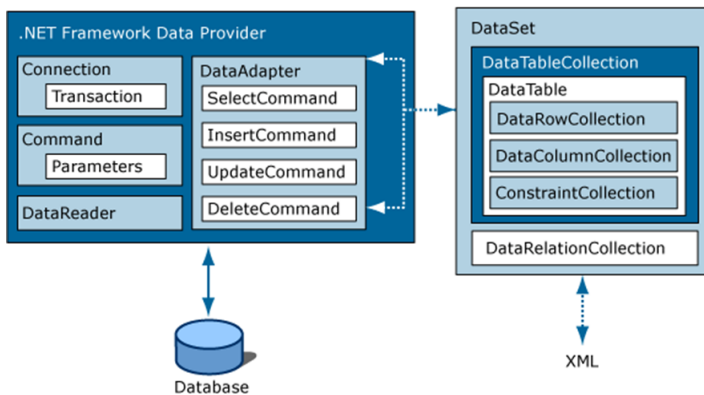



Fig. 6.1: Selecionando projeto para ter referência adicionada

Praticamente toda estrutura apresentada na figura 6.1 foi trabalhada no capítulo 5, no qual alguns dos recursos do .Net Framework Provider do SQL Server foram usados para implementar a aplicação de exemplo. Note no lado esquerdo da figura que, em um nível mais baixo, encontra-se uma base de dados que recebe e fornece dados para a estrutura de classes que os submetem e recebem. Ao lado direito da figura está o `DataSet` com a `DataTableCollection` e, dentro dela, a `DataTable`, que também foi apresentada no capítulo 5.

Como pode ser verificado pela figura 6.1, um `DataTable` pertence a uma `DataTableCollection`, da mesma maneira que uma tabela pertence a uma base de dados. Novamente no lado esquerdo da figura, o `DataAdapter` tem como componentes quatro `Commands`, cada um representando uma instrução SQL. Toda essa estrutura representa o `DataSet`, que tem como fonte de dados o `XML`, e será apresentada e trabalhada de maneira integrada neste capítulo.

6.2 Desenhando o formulário para a implementação de um exemplo de uso de DataSet e seus componentes

Como já dito e é possível de se verificar na figura 6.1, um `DataSet` é uma coleção de `DataTables`, pois possui dentro dele um objeto `DataTableCollection`. Para utilizar efetivamente um `DataSet`, é preciso possuir alguns `DataTables`.

Desta maneira, para implementar um exemplo, seguindo a lógica e problema apresentado no capítulo 5, será feito uso de dois `DataTables`: um para armazenar estados brasileiros e outro para armazenar as cidades de cada estado.

No Visual Studio, crie uma nova solução chamada `SolutionDataSet` e, dentro desta, um projeto `Windows Forms` chamado `DataSetProject`. Apague o formulário criado pelo template e crie um novo, chamado `FormDataSetTest`. Implemente seu formulário para que tenha a aparência do apresentado na figura a seguir:

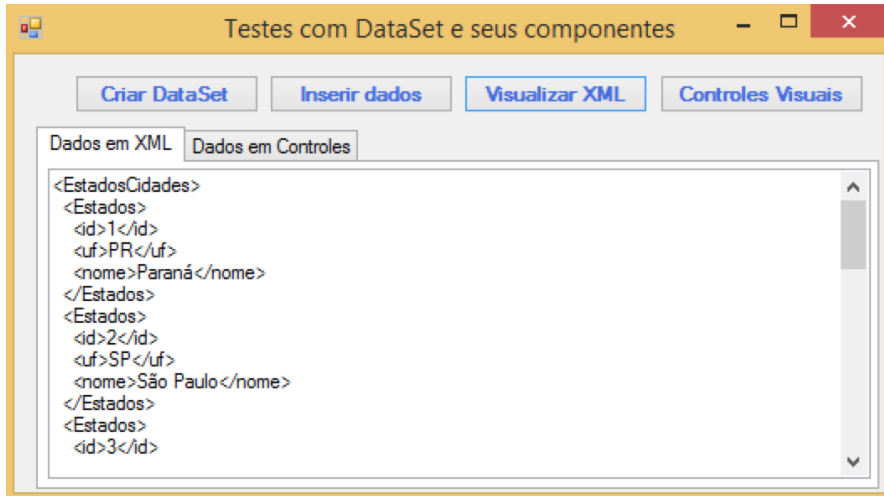



Fig. 6.2: Janela para interação e testes com DataSets

A parte superior do formulário é composta pelos botões que executarão as atividades relativas aos seus **captions**. Nos botões foi ajustada apenas a fonte do texto, para que seja de uma cor diferente e em negrito. Um novo controle foi inserido, o container `TabControl`, com duas guias (`Pages`). Na primeira guia – também chamada de Tab –, foi inserido um `TextBox` com a propriedade `MultiLine`, recebendo `True`, para assim aceitar múltiplas linhas, já que isso será necessário para a exibição do XML dos dados.

Na implementação que será apresentada na sequência, para que os dados XML apareçam, será preciso antes clicar no botão `Criar DataSet` e, em seguida, no `Inserir Dados`, para então clicar em `Visualizar XML`.

A segunda guia, que exibe os dados em controles visuais, pode ser visualizada na figura a seguir:

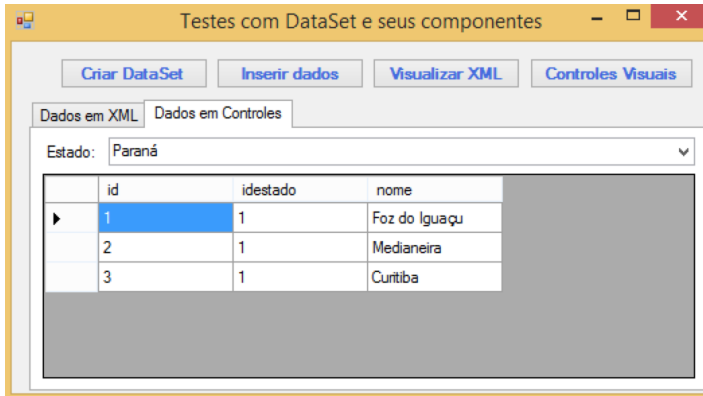



Fig. 6.3: Controles visuais para interação com DataSets

Na guia representada na figura 6.3, foi inserido o `TableLayoutPanel` com duas linhas e duas colunas, sendo que na primeira foram inseridos um `Label` e um `ComboBox` e, na segunda, um `DataGridView`. O tamanho da primeira coluna do `TableLayoutPanel` foi configurado para ser automático, desta maneira, ocupará a largura do `Label`. O `DataGridView` da segunda linha foi inserido na primeira célula e teve a propriedade `ColumnSpan` configurada com o valor 2. Para acessá-la, será preciso seguir a mesma ordem da apresentada para visualizar o XML.

Um `TabControl` contém `TabPages`, que são representadas por objetos `TabPage`, que podem ser inseridos por meio da propriedade `TabPages`. A ordem das `TabPages` na coleção do `TabControl` (`TabPages`) reflete diretamente na ordem em que as Tabs aparecerão no controle. O usuário poderá mudar a TabPage corrente clicando em qualquer uma das exibidas pelo controle. Diversas propriedades e métodos são oferecidos para esse controle, sendo uma das propriedades a `SelectedTab`, que permite atribuir ou obter a Tab atualmente em foco.

A visualização dos dados no exemplo dependerá da interação com o usuário. Ele precisará criar o DataSet, popular e depois visualizar. É interessante que, ao clicar em uma das guias de visualização, ele possa ver os dados, sem a necessidade de criar algo que deveria estar criado, uma vez que há a possibilidade de visualização. Duas são essas possibilidades:

1) Não exibir as guias de visualização até que os dados sejam disponibilizados;

2) Ao optar por visualizar os dados, caso eles não existam, eles devem ser criados automaticamente.

Pense nessas soluções e procure implementá-las para poder medir o que é mais produtivo.

6.3 Implementando a criação, população e interação com o DataSet

O primeiro botão da interface com o usuário é responsável por executar as instruções que criarão o DataSet e o disponibilizarão para a aplicação. De maneira análoga a uma base de dados, serão criadas as tabelas e seus relacionamentos, como é apresentado a seguir.

Resolução 6.1: método que implementa o evento Click do botão Criar DataSet

```
private void btnCriarDataSet_Click(object sender, EventArgs e){
    dsEstadosCidades = InitializeDataSet();
    MessageBox.Show("DataSet inicializado com sucesso");
}
```

O método `InitializeDataSet()` – representado na **Resolução 6.3** – cria um `DataSet` e seus componentes para a `DataTableCollection` e `DataRelationCollection`, e retorna o `DataSet` criado, que é atribuído ao campo `dsEstadosCidades`, declarado no início da classe. Veja a seguir.

Resolução 6.2: declaração do DataSet a ser utilizado

```
public partial class FormDataSetTest : Form {
    private DataSet dsEstadosCidades;

    public FormDataSetTest() {
        InitializeComponent();
```

```
    }
    // Código omitido
}
```

Resolução 6.3: método que inicializa o DataSet da aplicação

```
private DataSet InitializeDataSet() {
    DataTable dtEstados = new DataTable("Estados");
    dtEstados.Columns.Add("id");
    dtEstados.Columns.Add("uf");
    dtEstados.Columns.Add("nome");

    DataTable dtCidades = new DataTable("Cidades");
    dtCidades.Columns.Add("id");
    dtCidades.Columns.Add("idestado");
    dtCidades.Columns.Add("nome");

    DataSet dsEstadosCidades = new
    DataSet("EstadosCidades");
    dsEstadosCidades.Tables.Add(dtEstados);
    dsEstadosCidades.Tables.Add(dtCidades);

    DataRelation drCidadeEstado = new
    DataRelation("CidadesEstados",
    dtEstados.Columns["id"],
    dtCidades.Columns["idestado"]);
    dsEstadosCidades.Relations.Add(drCidadeEstado);

    return dsEstadosCidades;
}
```

Observe no código que, na definição dos `DataTables`, é enviado ao construtor o nome para a tabela que será criada. Uma vez criada a tabela, é preciso adicionar nela as colunas que a comporão. Veja que este processo é realizado pela invocação do método `Add()` da propriedade `Columns`, que é uma instância de `DataColumnCollection`.

Para ficar simples, é usado o método que recebe apenas o nome para a coluna, sendo que o tipo de dado será `string`. Existem outras duas sobre-

cargas para esse método, podendo definir a qual tipo de dado a coluna a ser adicionada se refere. Procure estudá-las e aplicá-las.

Criadas as duas tabelas, elas precisam ser adicionadas ao `DataSet`, para que o relacionamento entre elas possa ser configurado e adicionado ao `DataSet`. O relacionamento é estabelecido por meio de um objeto `DataRelation` que, em sua instanciação, recebe o nome para o relacionamento, a coluna pai (*Parent/Master Column*) e a coluna filho (*Child/Details Column*) e, então, é adicionado à propriedade `Relations` (`DataRelationCollection`) do `DataSet`.

Com o `DataSet` e as tabelas criados, é preciso agora populá-las, para que possam ser utilizadas. O segundo botão, `Inserir Dados`, tem sua implementação exibida na sequência.

Resolução 6.4: método que insere dados nos DataTables

```
private void btnInserirDados_Click(object sender, EventArgs e){
    DataTable dtEstados = dsEstadosCidades.
        Tables["Estados"];
    dtEstados.Rows.Add(1, "PR", "Paraná");
    dtEstados.Rows.Add(2, "SP", "São Paulo");
    dtEstados.Rows.Add(3, "SC", "Santa Catarina");

    DataTable dtCidades = dsEstadosCidades.
        Tables["Cidades"];
    dtCidades.Rows.Add(1, 1, "Foz do Iguaçu");
    dtCidades.Rows.Add(2, 1, "Medianeira");
    dtCidades.Rows.Add(3, 1, "Curitiba");
    dtCidades.Rows.Add(4, 2, "São Paulo");
    dtCidades.Rows.Add(5, 2, "Ilha Solteira");
    dtCidades.Rows.Add(6, 3, "Florianópolis");

    MessageBox.Show("Dados inseridos com sucesso.");
}
```

Observe que a inserção dos dados se dá por meio da invocação ao método `Add()`, pertencente à propriedade `Rows` (`DataRowCollection`). Verifique que a ordem dos parâmetros enviados corresponde à ordem em que as colu-

nas foram criadas. Note também que os valores para o atributo idestado são válidos, ou seja, respeitam a constraint da Relation existente entre as duas DataTables.

Agora que o DataSet já tem seus dados registrados, é possível trabalhar a visualização deles. Como foi verificado na figura 6.1, um DataSet registra seus dados em um arquivo XML, de maneira nativa. Desta maneira, o código que captura o evento para o clique do botão Visualizar XML é apresentado na sequência e é responsável pela exibição deles (figura 6.2).

Resolução 6.5: método que representa o evento Click do botão Visualizar XML

```
private void btnVisualizarXML_Click(object sender,
        EventArgs e) {
    tcResultados.SelectedTab = tpXML;
    txtXML.Text = dsEstadosCidades.GetXml();
}
```

O código determina que a TabPage referente ao XML seja exibida – note o nome que foi dado ao TabControl e a TabPage em questão – e, em seguida, o método que retorna o XML referente aos dados do DataSet é invocado e seu resultado exibido no TextBox.

O comportamento para o último botão é em respeito à exibição dos dados em controles visuais de interação direta com o usuário, e teve sua representação disponibilizada na figura 6.3. Veja seu código na sequência.

Resolução 6.6: método que representa o evento Click do botão Controles Visuais

```
private void btnVisualizarControes_Click(object sender,
        EventArgs e) {
    BindingSource bsMaster = new BindingSource();
    BindingSource bsDetails = new BindingSource();

    bsMaster.DataSource = dsEstadosCidades;
    bsMaster.DataMember = "Estados";
    bsDetails.DataSource = bsMaster;
```

```
    bsDetails.DataMember = "CidadesEstados";

    tcResultados.SelectedTab = tpComboEGrid;
    cbEstados.DataSource = bsMaster;
    cbEstados.DisplayMember = "nome";
    cbEstados.ValueMember = "id";
    dgCidades.DataSource = bsDetails;
}
```

É possível verificar na implementação desse método o uso de `BindingSource`s, que já foram usados anteriormente. Dois destes componentes são declarados: um responsável por apresentar os estados no `ComboBox` (tabela pai do relacionamento) e outro para apresentar as cidades no `DataGridView`.

Note que, na definição da fonte de dados (`DataSource`) do `BindingSource` de estados, é atribuído o `DataSet` que contém todas as tabelas. Desta maneira, é preciso configurar qual das tabelas será realmente utilizada. Isso pode ser feito atribuindo seu nome à propriedade `DataMember`. Como fonte de dados para `BindingSource` filho (`bsDetails`) – que conterá as cidades –, atribui-se o `BindingSource` pai da relação, o `bsMaster` – que contém os estados –, e o nome do relacionamento (o `DataRelation`) como `DataMember`.

Após a configuração dos `BindingSource`s, procede-se a configuração dos controles visuais. A do `DataGridView` é simples e já foi realizada. Para o `ComboBox`, como os dados não vêm de uma coleção de objetos de domínio, mas sim de strings, é preciso configurar qual valor é exibido (`DisplayMember`) e qual servirá como chave (`ValueMember`). Com essa configuração realizada, somada às do `BindingSource`s e da `DataRelation`, quando o usuário alterar um estado no `ComboBox`, o `DataGridView` exibirá apenas as cidades daquele estado.

Finalizando, como visto na aplicação anteriormente implementada e explicada, um `DataSet` permite que uma base de dados seja criada, populada e forneça também os dados nela armazenados. Essa base de dados, que fica na memória RAM, pode sofrer manutenções e interações, sem a necessidade de existência de um banco de dados para armazenamento, e os dados podem persistidos em um arquivo XML. Entretanto, o comum é que os dados

de uma base de dados estejam realmente em um banco de dados. Isso também é possível fazendo uso de DataSets, mais especificamente os `DataSets Tipados`.

6.4 DataSets Tipados

No capítulo 5, que fez uso de persistência em uma base dados, foi possível identificar uma dificuldade quando se usa uma base de dados relacional e uma aplicação orientada a objetos. Esta dificuldade refere-se ao mapeamento das informações fornecidas pelo usuário em sua interface com a aplicação (a janela de formulário), em objetos que representam a lógica de negócio (as classes) e ao mapeamento também desses objetos para um banco de dados relacional. A utilização dos objetos de negócio na camada intermediária – entre interface com o usuário e a base de dados – traz uma relativa perda de produtividade para a equipe de programação, aumentando, inclusive, a complexidade do código desenvolvido.

Para relembrar sobre essa dificuldade, na janela de manutenção dos dados de fornecedores, do capítulo 5, o usuário digita valores em `TextBoxs`, estes dados são mapeados para um objeto da classe `Fornecedor` e, em seguida, as propriedades desse objeto são obtidas para compor uma instrução SQL (`insert` ou `update`) em uma `string`. Para recuperá-los, uma instrução SQL `select` é também utilizada, seu resultado é mapeado para um objeto da classe `Fornecedor` e as suas propriedades são atribuídas para os `TextBoxs`.

É preciso também destacar que, além de todo o processo apontado anteriormente, existem situações de validação dos valores informados, que ainda não foram apresentadas nem trabalhadas. Um exemplo seria os valores de nome e CNPJ do fornecedor não serem vazios, e o CNPJ ser válido. Na nota de compra, uma validação seria a data de entrada não ser inferior à de emissão da nota, e a quantidade comprada não ser menor ou igual a zero.

Realizando uma análise nos problemas apresentados e com a tecnologia apresentada até o momento, conclui-se que essa situação deve-se ao fato de que os objetos de negócio não oferecem um meio para ligação direta com a interface do usuário. Isso também não ocorre com os dados obtidos de uma base de dados. Sendo assim, entende-se a necessidade de um objeto que

possa representar as tabelas, relacionamentos e as restrições da base de dados e que também possa ser fortemente tipado, permitindo seu uso também como objeto de negócio.

Com base nestes problemas, a Microsoft propõe uma arquitetura que utiliza o DataSet como um objeto de negócio, que representa as entidades da aplicação, pois, sendo ele independente de uma base de dados – como visto no exemplo introdutório deste capítulo –, pode atender bem às necessidades apresentadas e discutidas. Para isso, é preciso que o DataSet possua toda uma estrutura semelhante a uma base de dados, porém, conhecido como Schema XML, e que possa ser criado de maneira simplificada pelo Visual Studio.

6.5 Atualizando a base de dados

Para aplicar o conceito de DataSet Tipado, novas tabelas serão criadas. Buscando apresentar uma nova ferramenta, será criada uma base de dados diretamente no servidor do SQL Server Express e todo esse trabalho será realizado por meio dessa ferramenta, que é o *SQL Server Management Studio* (SSMS). Ela pode ser baixada de maneira isolada do site da Microsoft ou, dependendo da versão do SQL Server Express que você instalou, ela pode já estar disponibilizada em seu ambiente.

A figura 6.4 apresenta a janela inicial do **SSMS**:

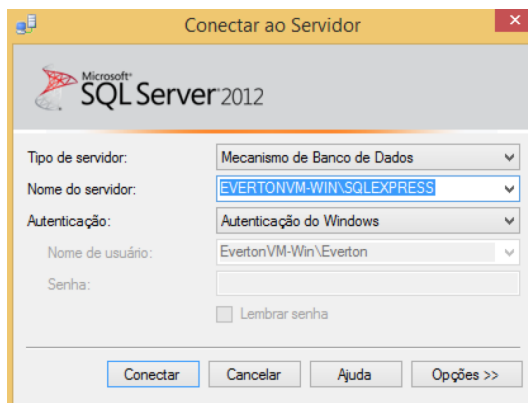



Fig. 6.4: Autenticação no SQL Server Management Studio

O nome do servidor informado na janela de autenticação é a composição do nome do computador em que o SQL Server Express foi instalado, com uma barra invertida (\) separando-o do nome da instância do banco, criada durante a instalação. `SQLEXPRESS` é a instância padrão, caso não tenha sido alterada. Se ao realizar a autenticação, ocorrer problemas de conexão, verifique se o serviço está iniciado, como já apresentado no capítulo 5.

Com a conexão realizada com sucesso, é preciso criar a base de dados para a aplicação. Todos os processos de criação e manutenção em objetos existentes no banco de dados (e em cada base de dados) podem ser realizados pela janela `Object Browser` (em português, essa janela foi traduzida para `Pesquisador de Objetos`), que é apresentada na figura a seguir.

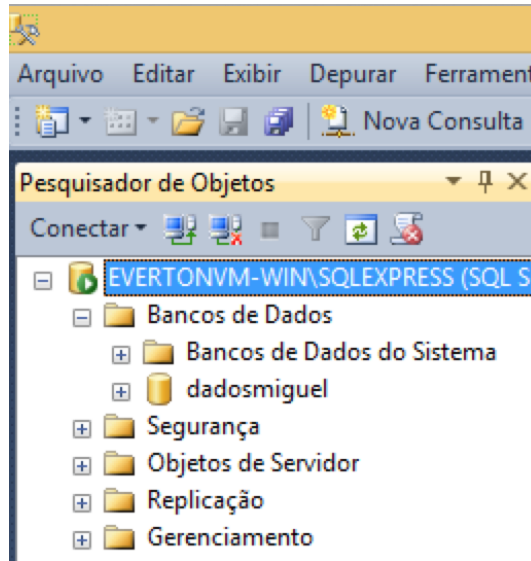



Fig. 6.5: Janela com os objetos existentes no banco de dados

Para criar a base de dados para a definição das tabelas que serão usadas, clique com o botão direito do mouse sobre a pasta `Banco de Dados` (que a tradução deveria ser base de dados) e, então, em `Novo Banco de Dados` (mesma observação quanto à tradução). Uma janela, semelhante à exibida na figura 6.6, será apresentada.

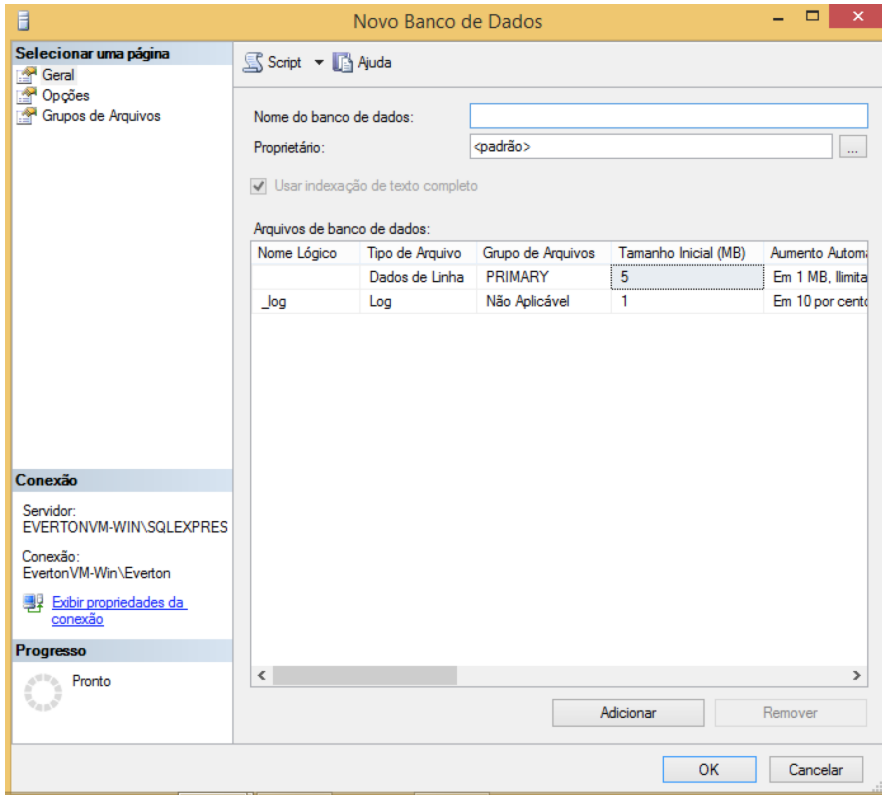



Fig. 6.6: Janela de definição de para a criação da base de dados

A janela da figura 6.6 possui diversas categorias e informações que podem ser preenchidas para a criação da base de dados. Entretanto, para o foco do livro, basta informar o nome dela, que será `LivroDataBase`. Após informar o nome, clique no botão `OK`, para a criação ser realizada.

Após a conclusão da criação da base se dados, é possível inserir as tabelas necessárias. Para isso, expanda a guia da base de dados criada e clique com o botão direito do mouse sobre `Tabelas` e, então, em `Nova Tabela`. A janela representada pela figura a seguir é exibida.

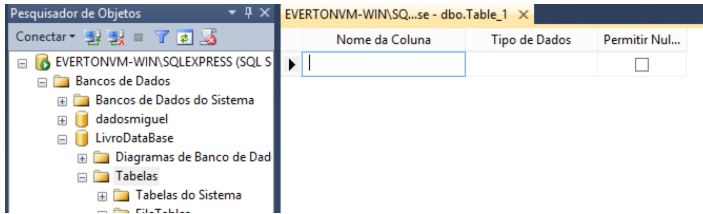



Fig. 6.7: Janela definição da estrutura de uma tabela

Assumindo que na nova aplicação que será criada a partir de agora serão necessárias informações referentes ao estado e cidade de um fornecedor, crie na janela da figura 6.7 a estrutura apresentada na figura seguinte:

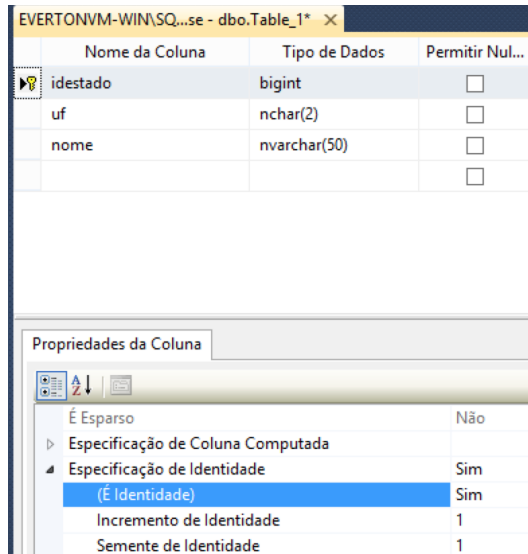
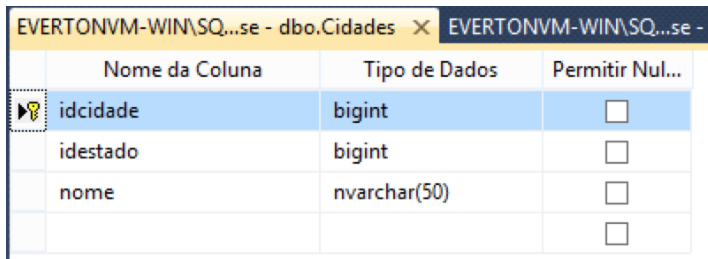



Fig. 6.8: Janela com a estrutura definida para a tabela Estados

Para que um campo seja definido como chave primária, no SSMS clique com o botão direito sobre o nome da coluna e clique em `Definir Chave Primária`. Depois, é preciso definir que o valor para a chave primária (o campo `idestado`) será autoincremento. Para isso, na janela de propriedades, abaixo da definição, expanda a propriedade `Especificação de`

`Identidade` e atribua `Sim` na subpropridade `(É Identidade)`, como pode ser verificado na figura 6.8. Com tudo definido, é preciso gravar a tabela. Clique no botão `Salvar`, abaixo do menu da janela, ou em `Arquivo` e `Salvar Tabela` ou, ainda, `Ctrl-S`. Um nome será solicitado. Informe este como `Estados`.

Na sequência, será preciso criar a tabela `Cidades`, de acordo com a estrutura apresentada na figura 6.9, criando o campo de chave primária, mas **não** definindo-o como identidade, no momento.

EVERTONVM-WIN\SQ...se - dbo.Cidades EVERTONVM-WIN\SQ...se -

Nome da Coluna	Tipo de Dados	Permitir Nul...
idcidade	bigint	☐
idestado	bigint	☐
nome	nvarchar(50)	☐
		☐

Fig. 6.9: Estrutura para a tabela Cidades

Uma vez criada e salva a tabela de cidades, altere a subpropriedade da chave primária para que seja autoincremento e salve a tabela. Se aparecer uma mensagem de erro, como apresentada na figura 6.10, será preciso realizar uma configuração no SSMS.

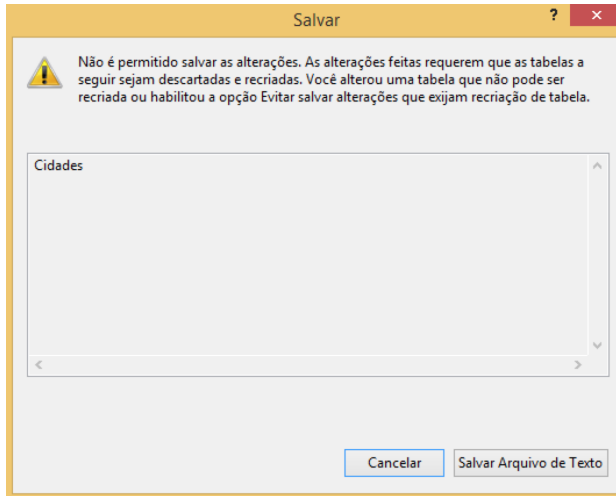



Fig. 6.10: Mensagem de alerta para a impossibilidade de alterar a estrutura da tabela

A mensagem de erro/alerta exibida na figura 6.10 refere-se à impossibilidade de atualizar a estrutura da tabela `Cidades`. Isso ocorre pelo fato de o campo de chave primária alterar a maneira de preenchimento de seu valor. Essa alteração pode causar perda de dados, por isso a impossibilidade. Entretanto, é possível configurar o SSMS para que realize essa atualização quando ocorrer esse tipo de problema. No menu do SSMS, acesse a opção `Ferramentas` e depois `Opções`. Na janela que será exibida, do lado esquerdo selecione a categoria `Designers` e desmarque a opção `Evitar alterações que exijam a recriação da tabela` (figura 6.11).

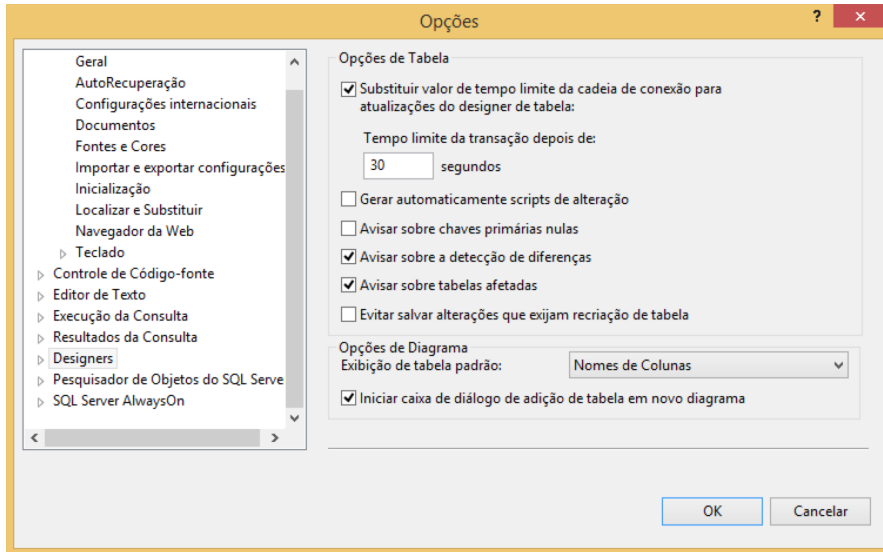



Fig. 6.11: Configuração para permitir alteração de tabelas que sofram modificações que causem a necessidade de recriar a tabela

Após realizada a configuração indicada, clique no botão `OK` da janela e proceda com a gravação da tabela, que ocorrerá perfeitamente.

É preciso agora implementar o relacionamento entre as tabelas `Estados` e `Cidades`. Esse relacionamento segue a ideia apresentada no capítulo 5. Entretanto, a implementação no SSMS é um pouco diferente. Clique com o botão direito do mouse sobre a área da estrutura da tabela e, no menu que aparece, selecione `Relacionamentos`. A janela representada pela figura a seguir será exibida.

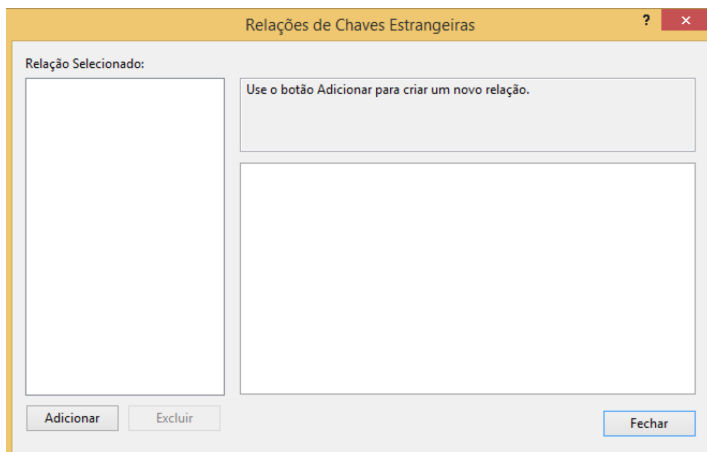



Fig. 6.12: Janela para adição de relacionamentos entre tabelas

Com a janela ativa, clique no botão `Adicionar`, para que o processo de configuração do relacionamento comece pela exibição da mesma janela, porém com propriedades que precisam ser configuradas (figura 6.13).

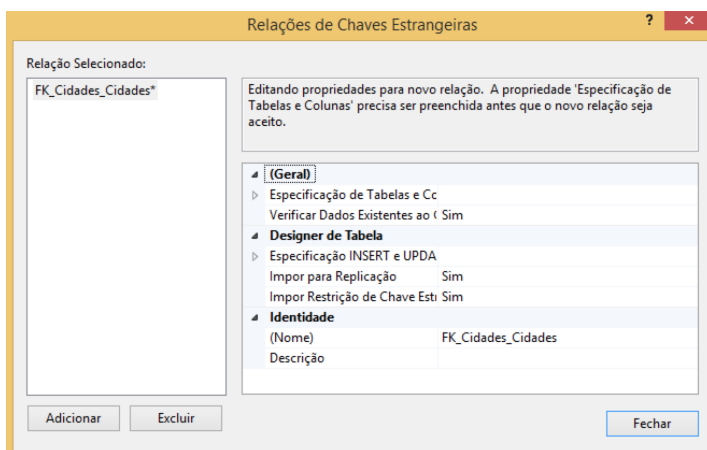



Fig. 6.13: Janela para configuração de relacionamentos

Nas propriedades que aparecem ao lado direito, clique na categoria

`Especificação de Tabelas e Colunas` e, em seguida, no botão que aparece ao lado direito. Com isso, uma janela para tal especificação será exibida (figura 6.14).

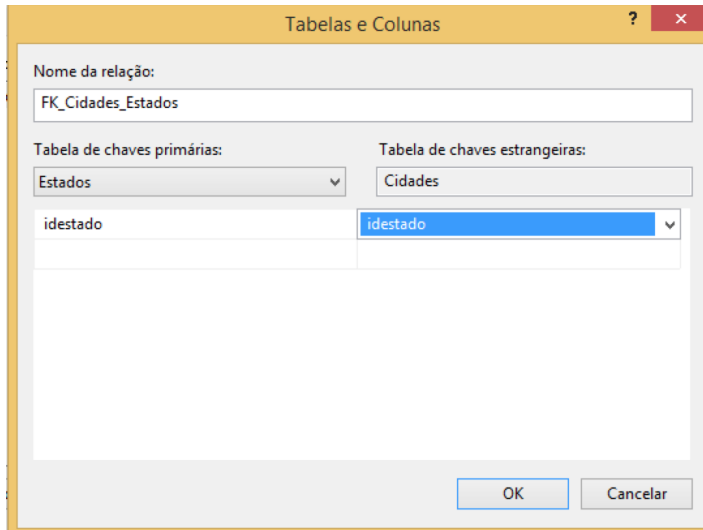



Fig. 6.14: Configuração das tabelas e colunas para o relacionamento a ser adicionado

Verifique na definição das tabelas e colunas que a tabela `Estados` é atribuída como primária, pois ela tem os dados de estados, que serão necessários na tabela `Cidades`. Desta maneira, a chave primária de `Estados` estará ligada com o campo `idestado` da tabela `Cidades`, tornando-se, então, uma chave estrangeira.

6.6 Criando os DataSets Tipados no Visual Studio

Crie uma nova solução, chamada `SolutionDataSetTipado`. Nela, crie um projeto `Windows Forms` chamado `DataSetTipadoProject`. Apague o formulário criado pelo template e crie no projeto uma pasta (*folder*) chamada `DataSets`. Clique no nome da pasta com o botão direito do mouse

e escolha a opção `Add` e `New Item`. Na categoria `Data`, escolha a opção `DataSet`, dê o nome de `DSEstadosECidades` para o arquivo e confirme a criação. Uma janela, representada pela figura a seguir, será exibida.

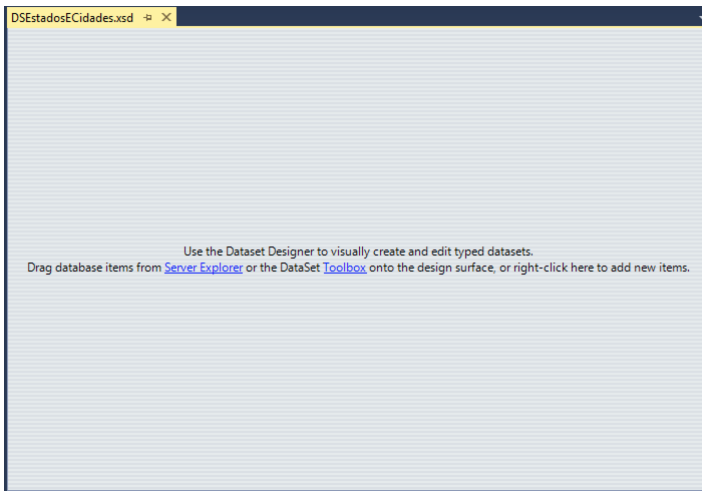



Fig. 6.15: Área de desenho do DataSet Tipado

A área de desenho para DataSets Tipados é semelhante à área de desenho de um formulário, no que diz respeito à versatilidade do padrão arrastar e soltar, embora os componentes possam ser inseridos e configurados de maneira isolada. O primeiro passo para a definição das tabelas que comporão este DataSet é disponibilizar no `Server Explorer`, os objetos da base de dados desejada. Para isso, na guia do `Server Explorer`, clique no botão `Connect to DataBase` (o segundo botão, da esquerda para a direita) e, então, a janela disposta na figura a seguir será exibida.

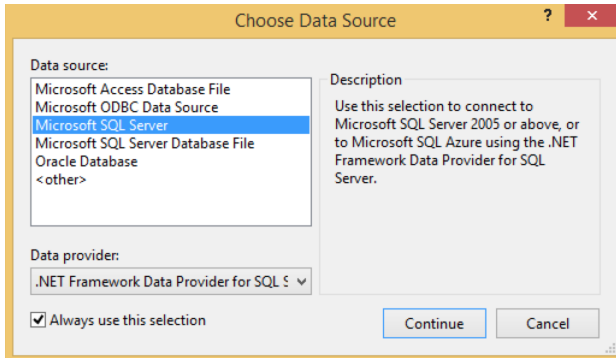



Fig. 6.16: Definindo a fonte de dados para a conexão

Note que, por definição, conexões com outros bancos de dados, além do SQL Server, estão disponibilizadas. Para continuar, selecione `Microsoft SQL Server`, como mostra a figura 6.16 e clique em continuar, para que a janela da etapa seguinte seja exibida (figura 6.17).

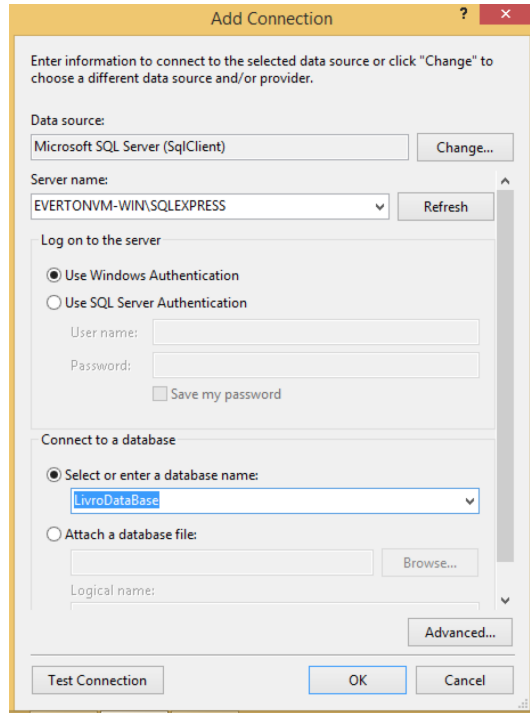



Fig. 6.17: Estabelecendo parâmetros para a conexão com a base de dados

Nessa janela (figura 6.17), é preciso informar o servidor (máquina) e a instância do SQL Server Express, da mesma maneira que foi informado para o acesso ao SSMS. Após isso, seleciona-se a base de dados que deseja conectar e, então, pode-se clicar em `Test Connection`, ou finalizar a configuração, clicando no botão `OK`. Com isso, a base de dados e todos seus objetos ficam disponibilizados no `Server Explorer`, como pode ser verificado por meio da figura 6.18.

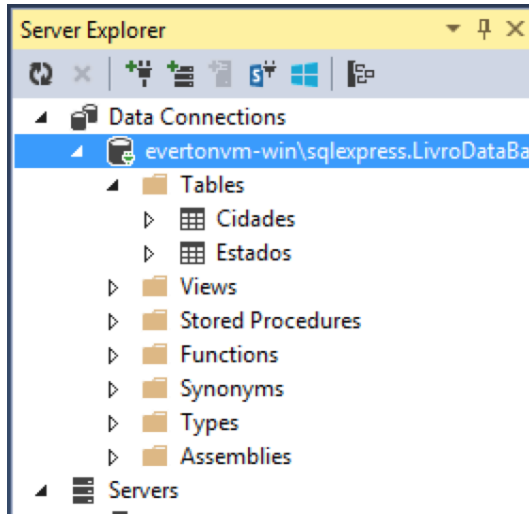



Fig. 6.18: Objetos disponibilizados no Server Explorer referentes à conexão criada

Agora, com a janela do DataSet Tipado visível no IDE, arraste as duas tabelas (`Estados` e `Cidades`) para a sua área de desenho. Com isso, o DataSet deverá estar semelhante ao apresentado na figura seguinte:

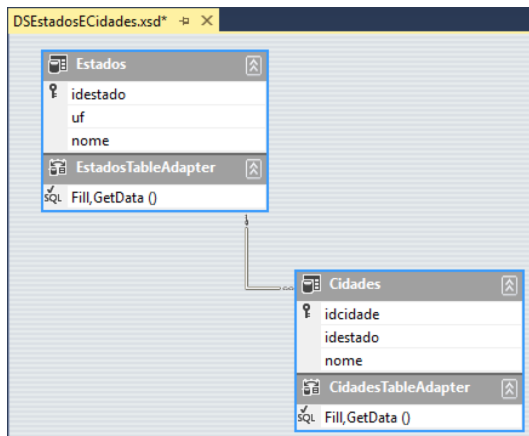



Fig. 6.19: DataSet Tipado criado com duas tabelas mapeadas

6.7 Entendendo os componentes do DataSet Tipado

Cada tabela mapeada da base de dados para o DataSet é composta por dois componentes: um `DataTable` e um `TableAdapter`. O `DataTable` representa a estrutura de campos da tabela, cujos nomes são exibidos na figura 6.19. Já o `TableAdapter` encapsula nele a interação com a base de dados, para atualizar e obter dados. Lembrando o trabalhado no capítulo 5, o `TableAdapter` faz uso de objetos `SqlConnection` e `SqlCommands`, como pode ser verificado na janela de propriedades do `TableAdapter` de `Estados`, o `EstadosTableAdapter` (figura 6.20).

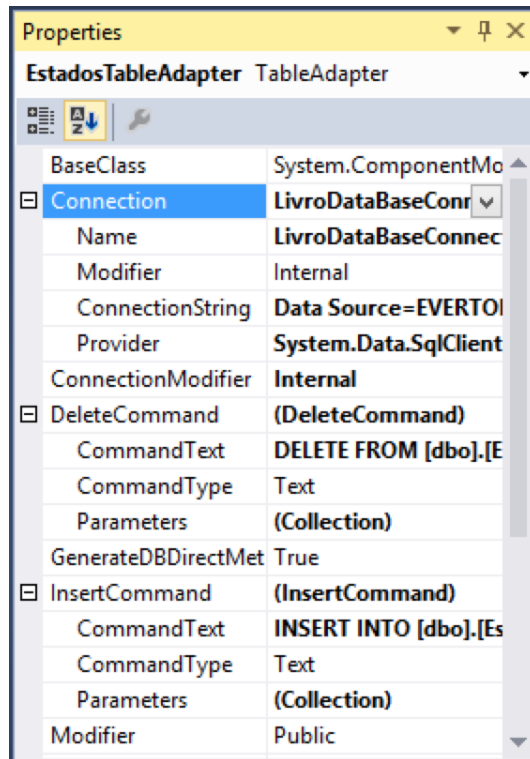



Fig. 6.20: Propriedades do TableAdapter de Fornecedores

Veja, na figura 6.20, que, para o objeto `EstadosTableAdapter`, existem configurações para as propriedades `Connection`, `DeleteCommand`, `InsertCommand` e as `SelectCommand` e `UpdateCommand`, que não aparecem na figura. Caso apareça apenas o `SelectCommand`, você não deve ter definido uma chave primária para a tabela. Para este momento, a recomendação é eliminar o arquivo de `DataSet`, atualizar a estrutura das tabelas e criar novamente o `DataSet`.

As subpropriedades de `Connection` estão definidas no arquivo `App.config`, que foi modificado quando a conexão com a base de dados foi criada. As instruções das propriedades `CommandText` foram definidas com base em todos os campos da tabela arrastada. Esses campos poderiam ser apenas alguns. Para o momento, procure não realizar modificações nas configurações criadas automaticamente.

As propriedades para o `DataTable` são apenas 3, dispensando a necessidade de uma figura. Entretanto, é interessante apresentar uma figura para as propriedades para uma coluna, e a figura 6.21 mostra as da coluna (`DataColumn`) `idestado`.

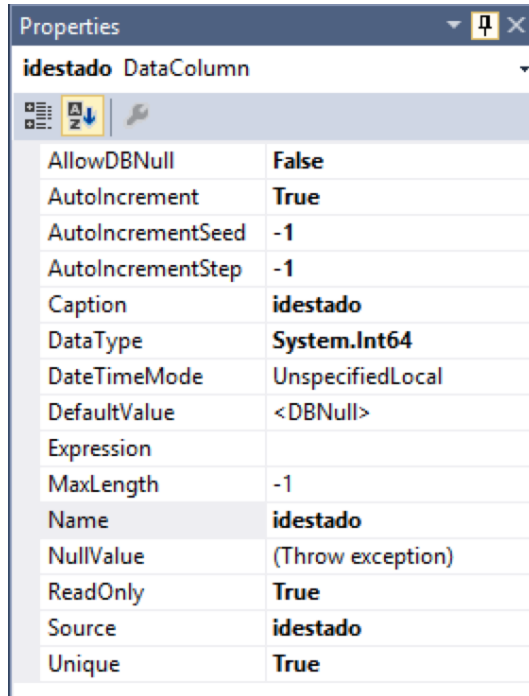



Fig. 6.21: Propriedades da coluna idestado

No DataSet Tipado, além de poder visualizar as tabelas mapeadas, é possível identificar que o relacionamento criado também foi mapeado como um `DataRelation`. Caso esse relacionamento não apareça, você se esqueceu de criá-lo na base de dados. Para este momento, a recomendação é eliminar o arquivo de `DataSet`, atualizar a estrutura das tabelas e criar novamente o `DataSet`.

6.8 Criando um formulário que faz uso de um DataSet Tipado

Na `Solution Explorer`, no projeto crie uma pasta chamada `Forms` e dentro dela outra chamada `CRUDs`. Dentro desta, crie um formulário chamado `FormEstados`. Com o novo formulário visível, abra a janela `Data`

`Sources` (Menu `View` → `Other Windows` → `Data Sources`). Sua janela deverá estar semelhante à apresentada pela figura na sequência.

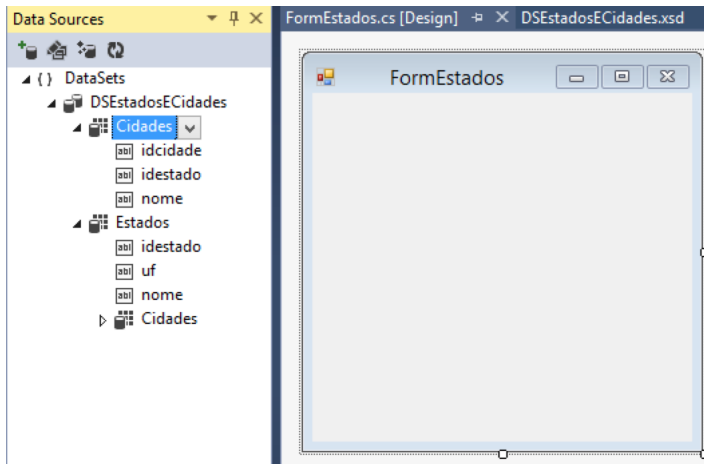



Fig. 6.22: Propriedades da coluna idestado

Uma observação extremamente importante é que, para realizar o que será apresentado agora, é preciso que o formulário que receberá os componentes do `DataSet` esteja visível. Com essa informação, note que o primeiro elemento da árvore exibida na janela é `DataSets`, que se refere à pasta onde o DataSet Tipado foi criado. Logo abaixo, encontra-se o próprio `DataSet` (`DSEstadosECidades`) e, dentro deste, elemento todos os `DataTables` nele existentes.

Para cada `DataTable`, uma relação de colunas é exibida. É possível verificar a ocorrência duplicada de `Cidades`, que, na realidade não é uma duplicidade. `Cidades`, que está dentro de `Estados`, na realidade representa o relacionamento entre as duas tabelas. Poderia-se dizer que `Cidades`, que está abaixo do `DataSet`, refere-se a todos os registros da tabela e, `Cidades`, que pertence a `Estados`, trará sempre apenas as cidades para o estado atual.

Os ícones ao lado de cada nome, tanto das tabelas como dos campos, possuem uma semântica, que está diretamente ligada a *como* os dados serão exibidos ao usuário, ou seja, quais controles devem ser criados no formulário. Como a janela que será criada é relacionada ao `CRUD` de `Estados`, é

importante que o template para os controles sigam essa ideia e que, como o `ID` de cada fornecedor é autoincremento, não há necessidade de ele ser um `TextBox`. Realize as alterações clicando no nome da tabela e no nome do campo, e escolha qual controle usará. Para a tabela `Estados`, escolha `Details` e para o `idestado`, escolha `Label`. Essa mudança pode ser verificada na figura seguinte.

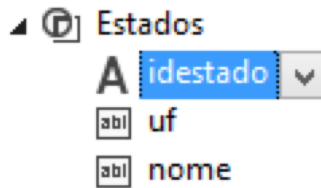



Fig. 6.23: Definição de controles visuais para a tabela de Estados

Com a definição concluída, clique no nome da tabela de `Estados` e a arraste para a área central do formulário. O resultado deve ser semelhante ao exibido na figura a seguir:

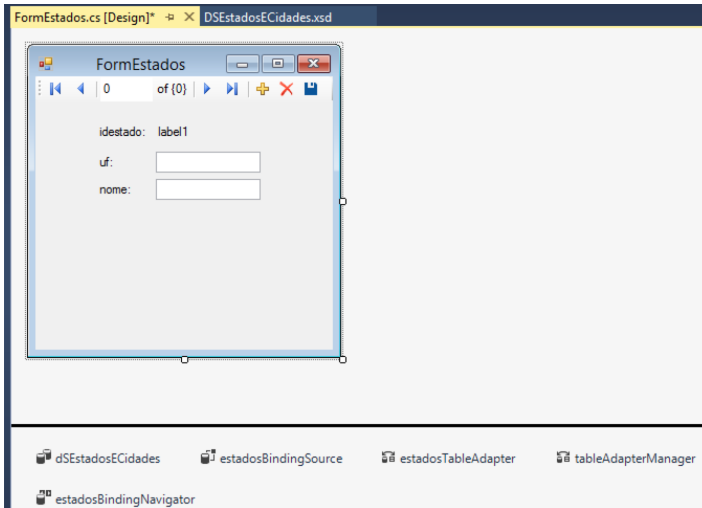



Fig. 6.24: Área de desenho do formulário com controles vinculados ao DataSet

Note na figura 6.24 que a área de desenho foi dividida em duas partes, a visual e não visual, que já havia sido trabalhada no capítulo 2. Na área de controles não visuais, note a existência de objetos referentes ao `DataSet`, ao `BindingSource` e ao `TableAdapter`. Esses controles já são conhecidos, entretanto, agora são utilizados de maneira visual, com maior produtividade. Dois controles novos são apresentados: o `TableAdapterManager`, que gerencia a funcionalidade de salvar dados relacionados, organizando a ordem de `inserts`, `updates` e `deletes`; e o `BindingNavigator`, que se refere à barra de navegação localizada no início da janela.

A ligação dos controles com o DataSet é realizada por meio da propriedade `DataBindings` de cada componente, que pode ser vista na figura a seguir:

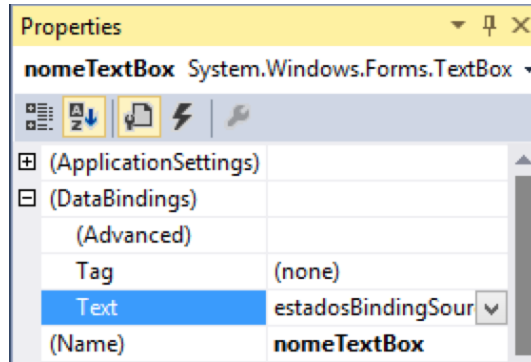



Fig. 6.25: Propriedade DataBindings dos controles visuais ligados ao DataSet

Verifique que a subpropriedade `Text`, de `DataBindings`, tem uma ligação com o `BindingSource`, pois é ele que fornece os dados vindos da base de dados e os envia para ela. Clique na `ComboBox` e veja como se dá esta configuração, representada a seguir pela figura:

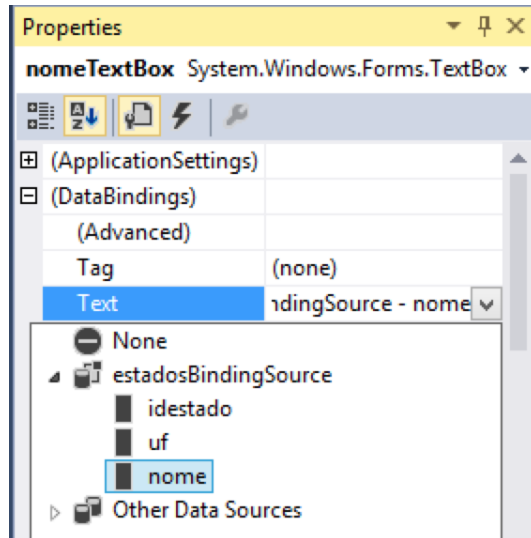



Fig. 6.26: Configuração da propriedade Text, de DataBindings

Verifique nas propriedades do objeto `estadosBindingSource` que o `DataSource` é o `DataSet`, e o `DataMember` é a `DataTable`, seguindo a mesma ideia do apresentado no início deste capítulo, porém, de maneira mais simples, ágil e visual (figura 6.27).

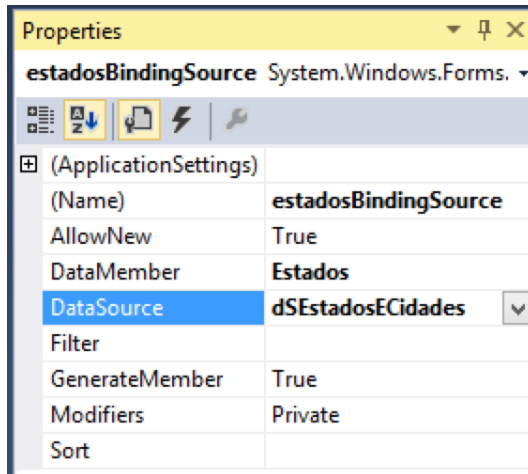
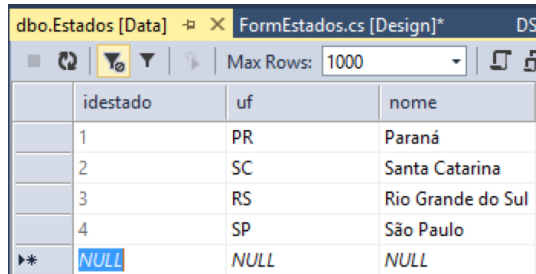



Fig. 6.27: Configuração da propriedade Text, de DataBindings

Para que alguns testes sejam realizados, insira alguns registros na tabela de `Estados` pelo `Server Explorer`, da mesma maneira que foi feito no capítulo 5. Se quiser, pode também inserir pelo SSMS, que tem a mesma funcionalidade. A figura 6.28 apresenta os dados inseridos por meio do `Server Explorer`.

dbo.Estados [Data] FormEstados.cs [Design]* DS

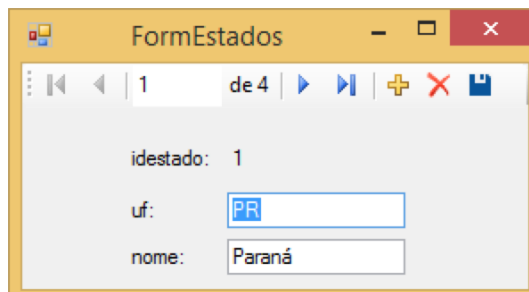
Max Rows: 1000

	idestado	uf	nome
	1	PR	Paraná
	2	SC	Santa Catarina
	3	RS	Rio Grande do Sul
	4	SP	São Paulo
▶*	NULL	NULL	NULL

Fig. 6.28: Dados para testes inseridos na tabela Estados

Altere a classe `Program`, para instanciar o formulário criado, e, então, execute a aplicação, que deverá apresentar um formulário semelhante ao exibido na figura 6.29. Note que na classe `Program` será necessário inserir o `using` referente ao `Namespace` do formulário, que é diferente de onde está essa classe.

Fig. 6.29: Janela para manutenção nos dados do fornecedor

Para facilitar os testes, crie uma janela com menus para acessarem os formulários que forem sendo criados ao decorrer deste capítulo.

6.9 Entendendo o comportamento do formulário em relação aos seus controles e ao DataSet

O primeiro ponto a ser explicado é em relação a como os dados da tabela já aparecem no formulário. Lembre-se de que o `DataSet` gera um objeto que reside na memória RAM do computador, ou seja, não tem vínculo direto de persistência com a base de dados. Desta maneira, para que o `DataSet` tenha os dados de tabelas mapeadas, é preciso populá-lo. Esta operação é realizada no momento em que o formulário é carregado (`Load`), como pode ser verificado na sequência.

Resolução 6.7: método que captura o evento Load do formulário

```
private void FormEstados_Load(object sender, EventArgs e) {
    // TODO: This line of code loads data into the
    // 'dSEstadosECidades.Estados' table. You can move, or
    // remove it, as needed.
    this.estadosTableAdapter.Fill(
    this.dSEstadosECidades.Estados);
}
```

Quando ocorreu o processo de arrastar os controles do `DataSource` para a área do formulário, o Visual Studio inseriu também alguns códigos, sendo um deles para o evento `Load` do formulário. A primeira parte é relacionada a um comentário de orientação sobre a linha inserida. Essa linha tem a chamada ao método `Fill()` do `adapter` de estados (`estadosTableAdapter`) e recebe como argumento o `DataTable` de estados (`estadosDataTable`), que pertence ao `DataSet`. Esse método é criado automaticamente pelo Visual Studio quando um mapeamento é criado na área de desenho do `DataSet`, e a sua invocação remete à execução de uma instrução `SELECT` na base de dados, que está inserida na propriedade `SelectCommand` do `TableAdapter` de estados. Esse método (`Fill()`) pode ser visualizado abaixo do `TableAdapter`, juntamente com o método `GetData()`, que retorna um `DataTable`.

A segunda funcionalidade a ser testada e explicada é em relação à inserção de novos dados. Execute a aplicação e clique no botão que adiciona um novo registro. Nesse novo registro inserido, veja que o `Id` recebe um valor negativo. Preencha os dados e clique no botão responsável pela gravação (os *hints* podem ser alterados normalmente por meio da janela de propriedades). Note que o `Id` passou a ser positivo, sendo exatamente o valor seguinte ao `Id` do último registro.

Verifique a propriedade `AutoIncrementSeed` da `DataColumn idestado`, na `DataTable` de `Estados` no `DataSet` criado. O valor é -1. Isso significa que sempre que os dados forem carregados (método `Fill()`), esse valor servirá de base. Verifique também a propriedade `AutoIncrementStep`, que é também -1. Com esta informação, a cada inserção de registro, o valor para o `Id` aumenta negativamente. Quando ocorre a gravação do registro, há uma atualização na base de dados, o que causa a real inserção dos dados nela. Veja que trago na sequência o código do evento `Click` do botão de gravação.

Resolução 6.8: método que captura o evento Click do botão de gravação

```
private void estadosBindingNavigatorSaveItem_Click(object
        sender, EventArgs e) {
    this.Validate();
    this.estadosBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(
        this.dSEstadosECidades);
}
```

A primeira instrução realiza um processo de validação no controle que tem o foco no momento da gravação. Essa validação se dá pelo disparo nos eventos `Validating` e `Validated`. A chamada ao método `EndEdit()` realiza atualizações pendentes no `DataSource` do controle. Finalizando, a chamada ao método `UpdateAll()` do `tableAdapterManager` invoca os métodos de atualizações dos `TableAdapters` existentes no formulário, enviando o `DataSet` onde as alterações físicas devem ocorrer. Neste momento, identifica-se o que foi feito localmente e os respectivos `commands` do

TableAdapter são invocados. Podem ser chamados os commands tanto para inserção, atualização ou exclusão, dependendo sempre do número e tipos de atualizações que foram realizadas localmente.

Para realizar um último teste, execute a aplicação, altere alguns dados, e exclua alguns registros e insira outros. Assim, feche a aplicação sem salvar e execute-a novamente. Veja que nenhuma das mudanças que fez foi persistida no banco. Em seguida, refaça as mesmas operações e grave antes de fechar. Execute novamente a aplicação e constate que todas as mudanças foram gravadas de uma única vez.

6.10 Criando um formulário utilizando dados de um relacionamento

O relacionamento entre as tabelas Estados e Cidades implica na existência de uma ou mais cidades para cada estado, e que cada cidade precisa de um estado. Desta maneira, na implementação da janela referente ao CRUD de cidades, será preciso que o usuário informe o estado de cada cidade a ser inserida, tendo como base os estados já armazenados.

Crie um novo formulário na pasta CRUDs, dentro da pasta Forms, e dê a ele o nome de FormCidades. Com o formulário visível, acesse a janela DataSources. Clicando no nome da DataTable Cidades, altere-a para Details. Altere a DataColumn idcidade para Label, e a idestado para ComboBox. Arraste a DataTable para a área de desenho do formulário. Na classe Program, defina o novo formulário para ser exibido e execute a aplicação (se criou a janela com menus, acesse por meio deles). A tela de sua aplicação deverá ser semelhante à apresentada na figura a seguir.

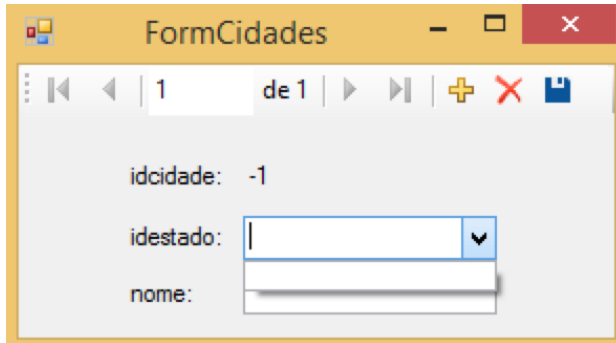



Fig. 6.30: Janela do cadastro de Cidades, com um ComboBox para selecionar estados

Observe na figura 6.30 que, ao inserir uma nova cidade e clicar no `ComboBox`, não aparecem os estados já gravados. Para configurar o `ComboBox` para exibir os estados já gravados, com a janela de cidades visível, vá à janela `DataSources`, clique na `DataTable` de estados, arraste-a até o `ComboBox` e solte-a. Após, clique na `Smart Tag` do `ComboBox` e sua janela deverá ser semelhante à apresentada na figura seguinte.

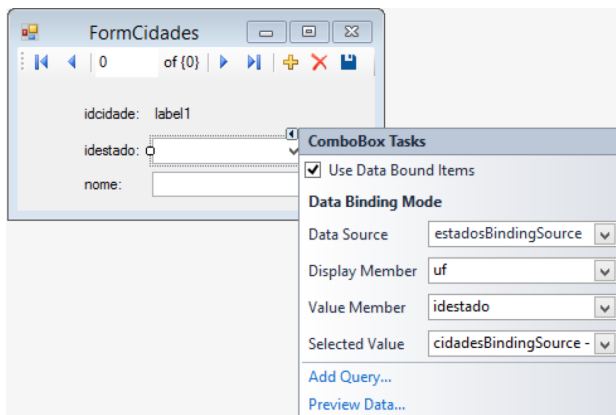



Fig. 6.31: Configuração do ComboBox

Quando houve o processo de "arrastar e soltar" (*drag and drop*) do

DataTable de estados para o ComboBox, o Visual Studio realizou a configuração básica do componente. Primeiramente, a ferramenta marcou a opção Use Data Bound Items, que informa ao controle que os dados fornecidos por ele serão ligados a uma fonte de dados, e não digitados de maneira estática. Em seguida, configurou-se a fonte de dados, o Data Source.

Quando o DataTable de cidades foi arrastado, foi criado para ele um BindingSource como fonte de dados para cidades. A figura 6.32 mostra a seleção do BindingSource, se fosse realizada de maneira manual.

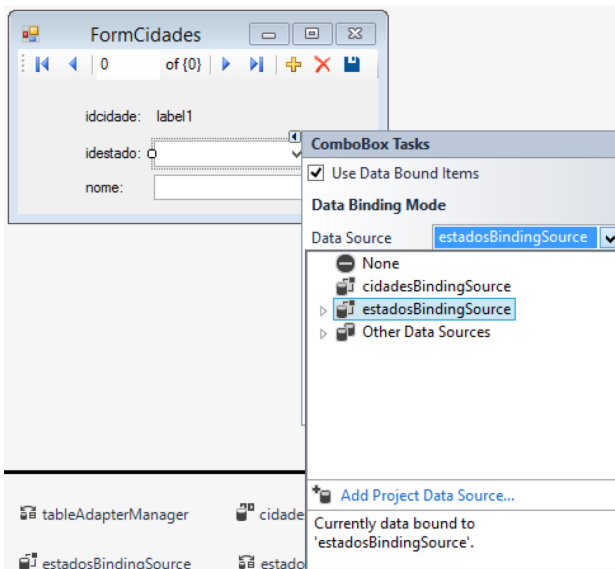



Fig. 6.32: Verificando a seleção de maneira manual do BindingSource para um ComboBox

Todos os objetos BindingSources que constam nesta janela de configuração existem no formulário ativo. Em Other Data Sources, é possível obter objetos que existem no projeto. Selecionando um BindingSource do projeto, um objeto referente a ele é criado no formulário ativo.

A ComboBox referente ao Display Member refere-se à seleção de qual DataColumn fornecerá o dado que será exibido na lista de itens do controle. Veja que a configuração feita pelo arrastar e soltar definiu a UF, mas o correto

seria `NOME`. Proceda com a correção.

A `ComboBox` referente ao `Value Member` trata da `DataColumn` que tem o valor referente à chave primária do dado selecionado pelo usuário. Quanto à `ComboBox Selected Value`, ela refere-se à `DataColumn` da `DataTable` que receberá o valor selecionado, que representa a chave estrangeira, mantendo a associação entre os objetos.

Com toda a configuração realizada e explicada, execute agora a aplicação, insira um novo registro e selecione um estado para a nova cidade (figura 6.33).

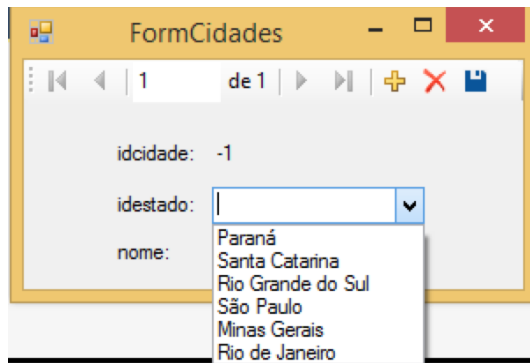



Fig. 6.33: Visualizando os estados previamente cadastrados para seleção em um ComboBox

Como é possível verificar, os nomes dos estados apresentados para seleção no `ComboBox` não estão em uma ordem alfabética, o que pode dificultar a busca por parte do usuário. O primeiro ponto para realizar esta mudança é saber onde está ocorrendo a recuperação dos dados que populam o `ComboBox` (visto na sequência).

Resolução 6.9: método Load que popula os DataTables do formulário

```
private void FormCidades_Load(object sender, EventArgs e) {
    this.estadosTableAdapter.Fill(this.dSEstadosECidades.
        Estados);
    this.cidadesTableAdapter.Fill(this.dSEstadosECidades.
```

```
        Cidades);
}
```

Observe que, além da chamada ao método `Fill()` enviando o `DataTable Cidades` como argumento, há agora uma nova chamada ao mesmo método, porém enviando o `DataTable` de estados. Identificando como o `DataTable` é populado, é preciso adaptar o método, ou criar outro que carregue os registros com base em uma classificação, que neste caso é o nome dos estados. No `DataSet`, clique com o botão direito do mouse sobre os nomes dos métodos abaixo do `TableAdapter` de `Estados` (figura 6.34).

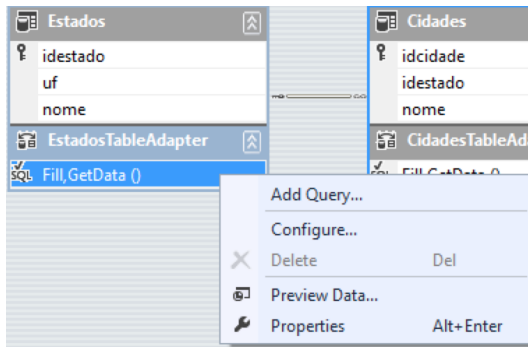



Fig. 6.34: Menu de contexto do TableAdapter

O menu de contexto para o `TableAdapter` possui uma opção para criação de novos métodos, que é a `Add Query...`. Confirme essa opção e a janela para criação de uma nova consulta é apresentada (figura 6.35).

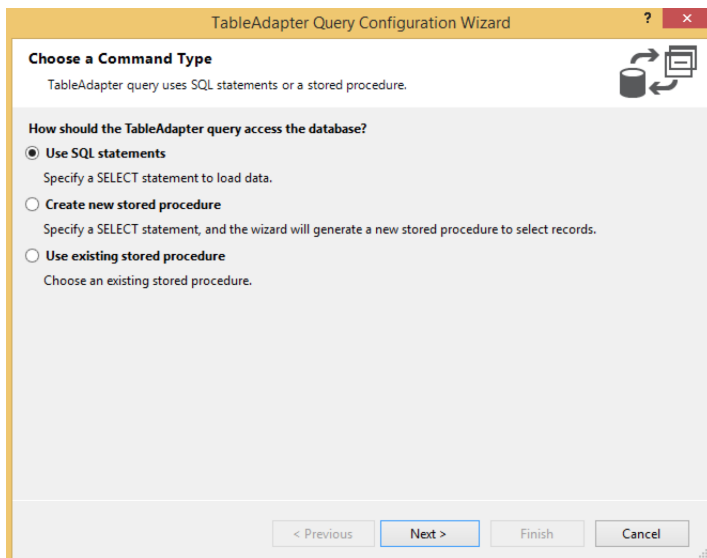



Fig. 6.35: Janela para seleção do tipo de Command que será criado

A consulta que será criada é semelhante à criada automaticamente para o método `Fill()`, e ela faz uso da instrução `SELECT` da SQL. Desta maneira, marque essa opção (que é a padrão) e clique no botão `Next`. Assim, uma nova janela aparece (figura 6.36).

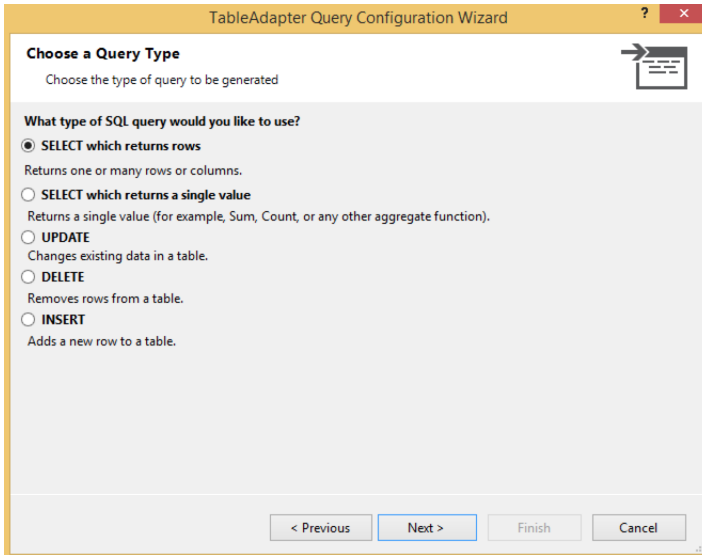



Fig. 6.36: Seleção do tipo de instrução a ser executada pelo Command

Com a opção padrão marcada, que é um `SELECT que retorna muitas linhas e/ou colunas`, clique no botão `Next` e uma nova janela é apresentada (Figura 6.37).

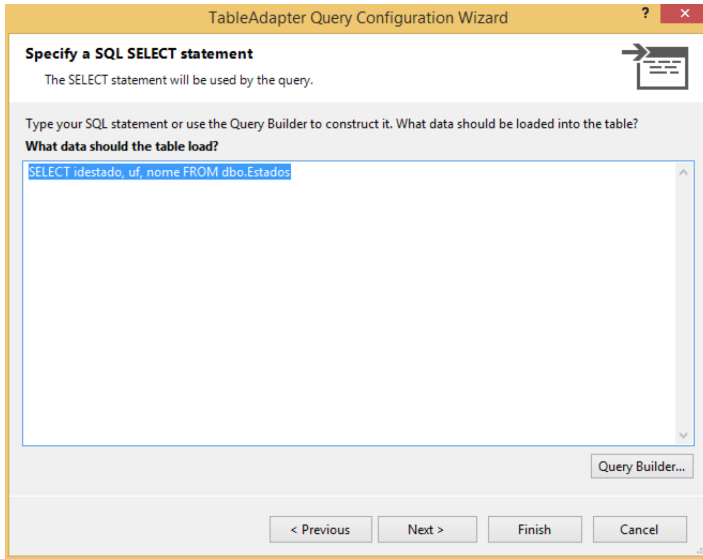



Fig. 6.37: Instrução que será executada pelo método do Command

Na instrução SQL apresentada, é possível realizar adaptações da maneira que desejar ou precisar. É importante saber que não se devem mudar os campos que serão selecionados, pois o retorno será para o `DataTable` já existente. Insira ao final da instrução a cláusula " `ORDER by nome`". Caso seja desejável toda a criação da instrução, pode ser realizada pela janela que se exibe quando se pressiona o `Query Builder...`. Concluindo a mudança, pressione o botão `Next` e a janela da figura 6.38 é apresentada.

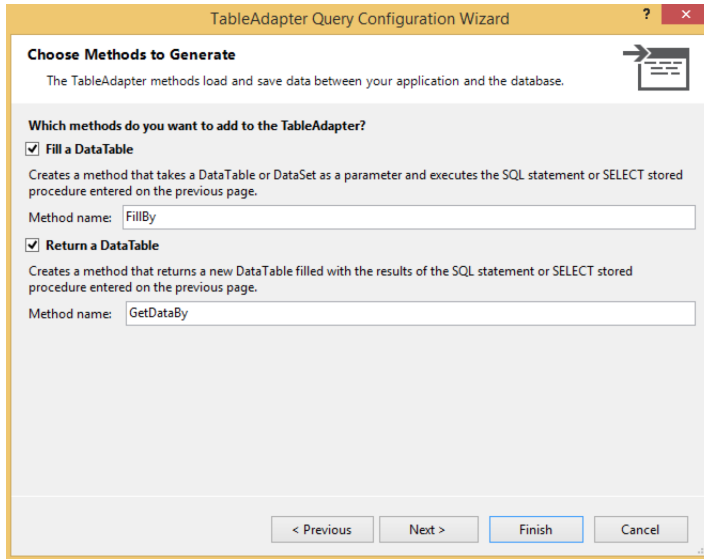
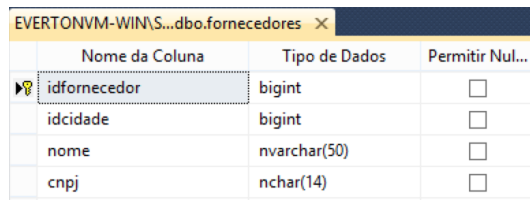
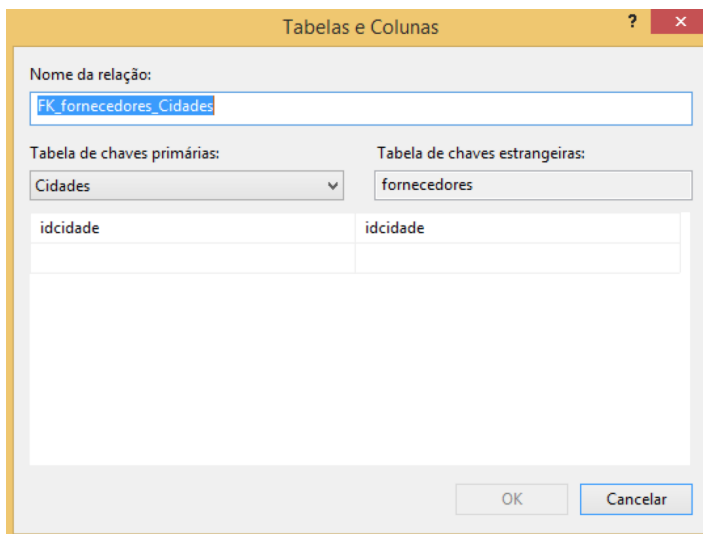



Fig. 6.38: Definindo os nomes para os métodos que serão criados no TableAdapter

Nos nomes que aparecem como sugestão para cada método, existe o sufixo `By`; complete o nome com o do campo que classifica a seleção. Neste caso, o método que popula uma `DataTable` deverá ser chamado de `FillByNome()`. Desmarque a opção do método que retorna uma `DataTable`, ela não será necessária. Neste momento, você pode clicar diretamente no botão `Finish`. Verifique agora que, nos métodos do `TableAdapter` de `Estados`, consta o novo método criado. Adapte, no método que captura o evento `Load` do formulário, para que, em vez de chamar o método `Fill()`, seja agora chamado o método `FillByNome()`. Execute a aplicação, insira uma nova cidade e, ao selecionar o estado, veja que agora aparecem em ordem alfabética.

6.11 Concluindo os formulários para a aplicação de compra e venda

Com base no que foi apresentado neste capítulo, algumas tabelas serão criadas ou modificadas, tendo sempre como exemplo a base de dados do capítulo 5. Como primeira atividade, crie a tabela `Fornecedores` (figura 6.39). Essa tabela, neste exemplo conterá um relacionamento (figura 6.40) com a tabela `Cidades`. Lembre-se de configurar a chave primária para ser identidade (autoincremento).

Fig. 6.39: Estrutura para a tabela Fornecedores

Fig. 6.40: Relacionamento entre as tabelas Fornecedores e Cidades

Com a tabela criada, é preciso agora levá-la para a aplicação. Deixe abertas as janelas do `DataSet` e do `Server Explorer`, encontre a tabela e a arraste para o `DataSet` (figura 6.41).

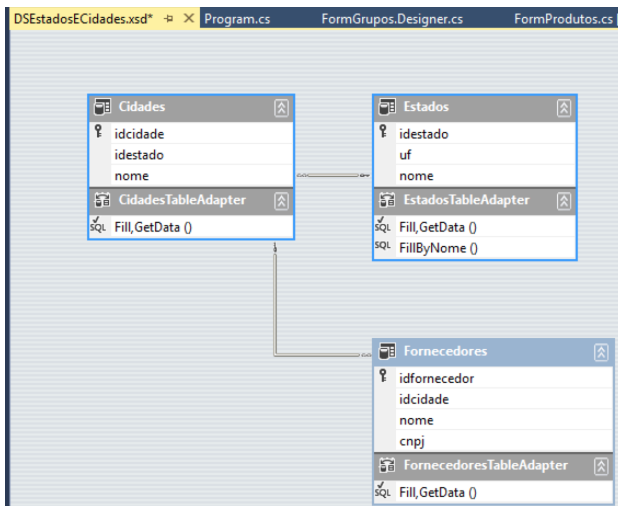



Fig. 6.41: DataSet com a nova tabela de Fornecedores e o relacionamento com Cidades

No `TableAdapter` da tabela de cidades, adicione um novo método, chamado `FillByNomeComUF()` (figura 6.42). Ele será usado para exibir o nome da cidade no `ComboBox` da `Cidade`, na janela de `Fornecedores`, seguida da UF do estado a que ela pertence, como: `Foz do Iguaçu (PR)`.

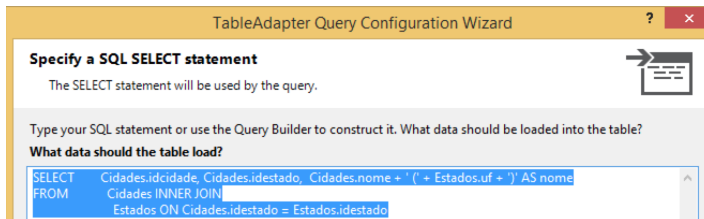



Fig. 6.42: Instrução SELECT para obtenção do nome da cidade com a UF de origem

Como, ao lado do nome da cidade, é preciso obter a UF referente a ela, e esta informação não está na tabela de cidades – foco principal do `SELECT` –, é preciso trazer para o `SELECT` a tabela de estados também. Esta operação é feita pela cláusula `INNER JOIN`, na qual ela recebe como argumento a chave estrangeira que será ligada à chave primária da tabela do `INNER JOIN`.

As associações internas (ou `INNER JOIN`) combinam tabelas comparando valores em colunas que sejam comuns a ambas as tabelas. O resultado para uma consulta, fazendo uso dessa cláusula, retorna somente registros que correspondam às condições da associação. Desta maneira, caso existam registros que, nas colunas especificadas, não correspondam à operação, eles não farão parte do resultado obtido.

Caso deseje, o `INNER JOIN`, ou qualquer tipo de relacionamento ou seleção de dados a serem retornados, pode ser implementado por meio da janela `Query Builder`. Para isso, clique no botão que a exibe e a janela da figura 6.43 é apresentada.

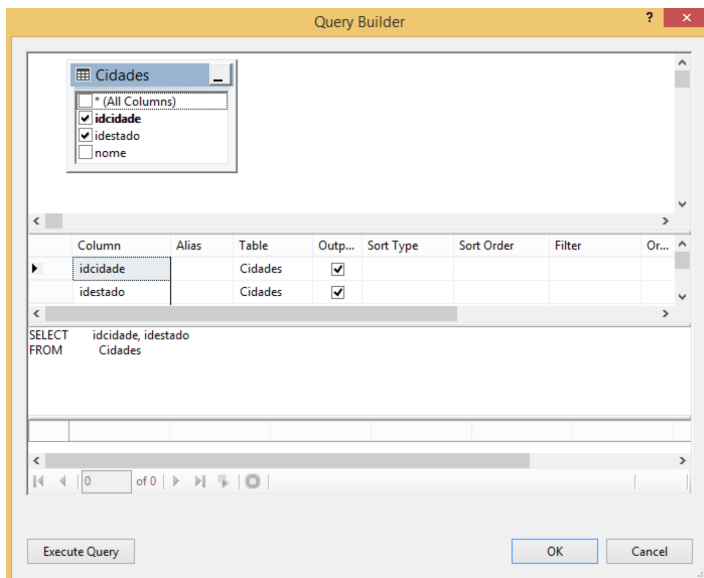



Fig. 6.43: Janela para criação de consultas de maneira visual – Query Builder

Por meio da janela `Query Builder`, é possível alterar diretamente a

instrução SQL ou fazer uso do recurso visual, característico em aplicações Windows. Clique com o botão direito do mouse sobre a área em que aparece o desenho da tabela `Cidades`, selecione `Add Table`, escolha a tabela `Estados`, clique no botão `Add` e, então, em `Close` (figura 6.44).

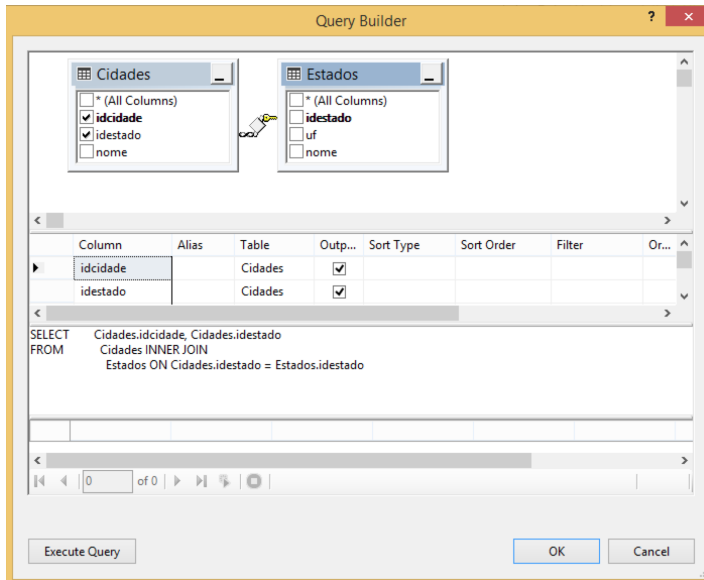



Fig. 6.44: Formulário de Fornecedores com ComboBox personalizado para seleção da Cidade

Na nova tabela que apareceu, selecione a coluna `UF`. Adapte a instrução `SELECT` para que fique igual a apresentada anteriormente, na figura 6.42. Clique no botão `OK` e depois em `Finish` para finalizar.

Agora, para criar a interface de interação com o usuário (a janela), crie um novo formulário para `Fornecedores` – na pasta `CRUDs` – (figura 6.45), seguindo a mesma situação apresentada para o formulário de `Cidades`. No `ComboBox` de `Cidade`, arraste, da janela `Data Sources`, o `DataTable` de `Cidades`. Adapte o método do evento `Load` para chamar o método anteriormente criado (visto na sequência).

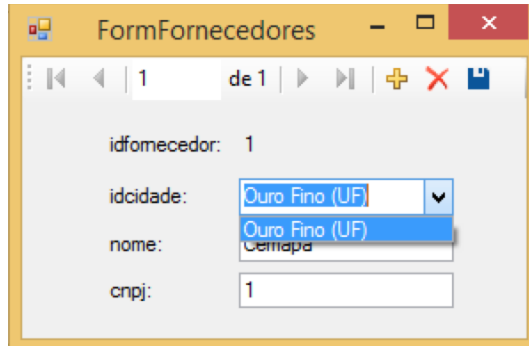
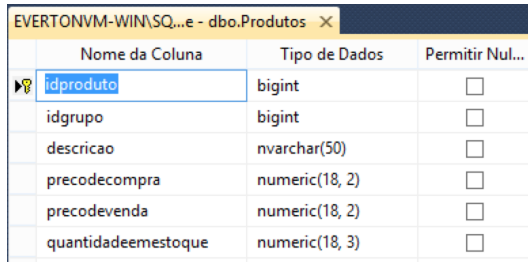



Fig. 6.45: Formulário de Fornecedores com ComboBox personalizado para seleção da Cidade

Resolução 6.10: método Load que popula os DataTables do formulário

```
private void FormFornecedores_Load(object sender, EventArgs e){
    this.cidadesTableAdapter.FillByNomeComUF(this.
        dSEstadosECidades.Cidades);
    this.fornecedoresTableAdapter.Fill(this.
        dSEstadosECidades.Fornecedores);
}
```

Com a intenção de praticar mais a implementação de relacionamentos, a aplicação deste capítulo trará para `Produtos` a categorização em grupos. Desta maneira, crie uma tabela de `Grupos` (figura 6.46) e uma de `Produtos` (figura 6.47), e atribua a elas um relacionamento (figura 6.48). Novamente, lembre-se de atribuir a propriedade de identidade às chaves primárias.

EVERTONVM-WIN\S...ase - dbo.Grupos

Nome da Coluna	Tipo de Dados	Permitir Nul...
idgrupo	bigint	☐
nome	nvarchar(50)	☐

Fig. 6.46: Estrutura para a tabela de Grupos de Produtos

EVERTONVM-WIN\SQ...e - dbo.Produtos

Nome da Coluna	Tipo de Dados	Permitir Nul...
idproduto	bigint	☐
idgrupo	bigint	☐
descricao	nvarchar(50)	☐
precodecompra	numeric(18, 2)	☐
precodevenda	numeric(18, 2)	☐
quantidadeemestoque	numeric(18, 3)	☐

Fig. 6.47: Estrutura para a tabela de Produtos

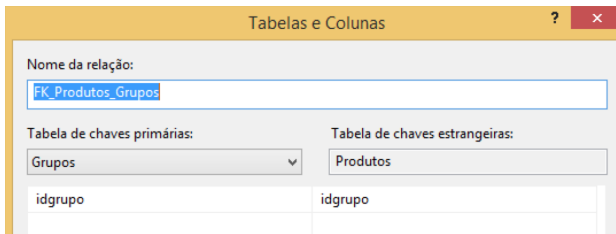



Fig. 6.48: Relacionamento entre as tabelas de Grupos e Produtos

Com as tabelas e seus relacionamentos criados, o passo seguinte é levar para a aplicação essas tabelas. Desta maneira, deixe as janelas do `DataSet` e `Server Explorer` ativas, e arraste-as para a área de desenho do `DataSet`. No `TableAdapter` de `Grupos`, crie um novo método para classificar a seleção por ordem do nome do grupo, tal qual foi feito para `Cidades` (figura 6.49).

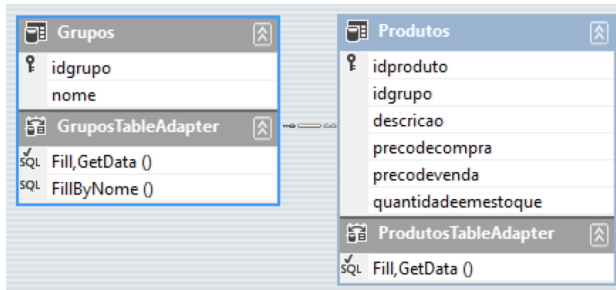



Fig. 6.49: Tabelas e relacionamentos de Grupos e Produtos inseridos no DataSet, com o método FillByNome() já criado

O passo final para o trabalho referente a `Grupos` e `Produtos` é a criação dos respectivos formulários (figuras 6.50 e 6.51). Crie um formulário para cada tabela, ative a janela `Data Sources`, configure os componentes e arraste-os para a área de desenho. Após isso, na janela de `Produtos`, adapte o método que captura o evento `Load` do formulário, para que faça uso do novo método de população do `DataTable` de `Grupos`, que preencherá o `ComboBox`. Veja isso na sequência.

Fig. 6.50: Formulário para manutenção e registros de grupos de produtos

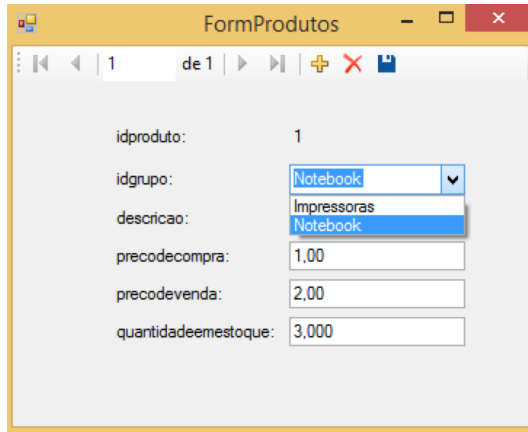



Fig. 6.51: Formulário para manutenção e registros de produtos

Resolução 6.11: método Load que popula os DataTables do formulário

```
private void FormProdutos_Load(object sender, EventArgs e) {
    this.gruposTableAdapter.FillByNome(this.
        dSEstadosECidades.Grupos);
    this.produtosTableAdapter.Fill(this.
        dSEstadosECidades.Produtos);
}
```

Como um adicional para este capítulo, a aplicação proposta oferecerá também a janela para vendas e não apenas entrada (compra) de estoque. Desta maneira, faz-se necessária uma tabela de `Clientes` (figura 6.52), com sua respectiva disponibilização no `DataSet` (figura 6.53), e o formulário para interação com o usuário (figura 6.54). Atualize o método do evento `Load` para obter os dados do método que populará o `ComboBox` de cidades, criado para o `TableAdapter` de `Cidades`. Veja o código a seguir.

Fig. 6.52: Estrutura para a tabela de Clientes

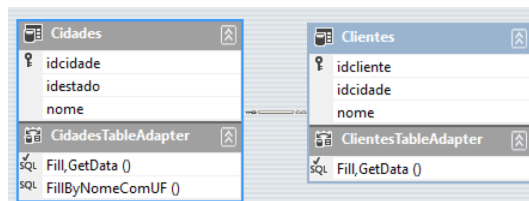



Fig. 6.53: DataTable de Clientes com DataRelation para Cidades

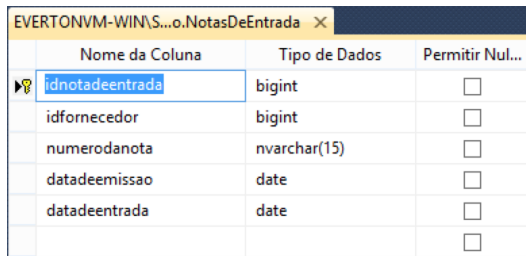



Fig. 6.54: Formulário para manutenção dos dados e registro de Clientes

Resolução 6.12: método Load que popula os DataTables do formulário

```
private void FormClientes_Load(object sender, EventArgs e) {
    this.cidadesTableAdapter.FillByNomeComUF(this.
        dSEstadosECidades.Cidades);
    this.clientesTableAdapter.Fill(this.
```

```
        dSEstadosECidades.Clientes);
}
```

A janela para o registro de uma nota de entrada e seus produtos será aperfeiçoada neste capítulo, fazendo uso dos recursos oferecidos pelo `DataSet`. O primeiro passo é a criação das tabelas de `NotasdeEntrada` (figura 6.55) e de `ProdutosNotasdeEntradas` (figura 6.57), e seus respectivos relacionamentos (figura 6.56, 6.58 e 6.59).

EVERTONVM-WIN\S...o.NotasDeEntrada

Nome da Coluna	Tipo de Dados	Permitir Nul...
idnotadeentrada	bigint	☐
idfornecedor	bigint	☐
numerodanota	nvarchar(15)	☐
datadeemissao	date	☐
datadeentrada	date	☐
		☐

Fig. 6.55: Estrutura para a tabela de Notas de Entrada

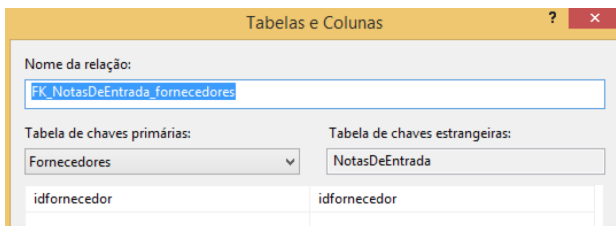
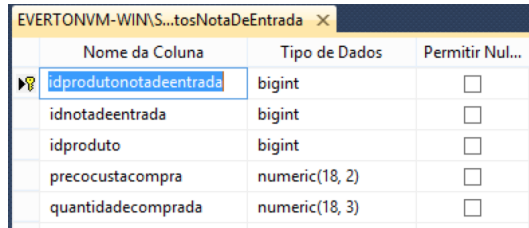



Fig. 6.56: Relacionamento entre as tabelas de Notas de Entrada e Fornecedores

EVERTONVM-WIN\S...tosNotaDeEntrada

Nome da Coluna	Tipo de Dados	Permitir Nul...
idprodutonotadeentrada	bigint	☐
idnotadeentrada	bigint	☐
idproduto	bigint	☐
precocustacompra	numeric(18, 2)	☐
quantidadecomprada	numeric(18, 3)	☐

Fig. 6.57: Estrutura da tabela de Produtos para Notas de Entrada

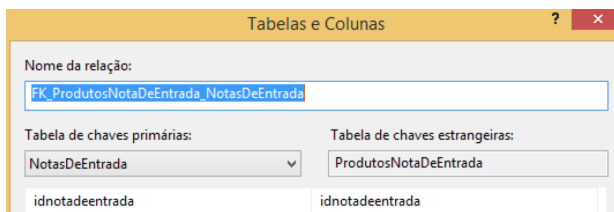



Fig. 6.58: Relacionamento entre as tabelas de Produtos de Notas de Entrada e Notas de Entrada

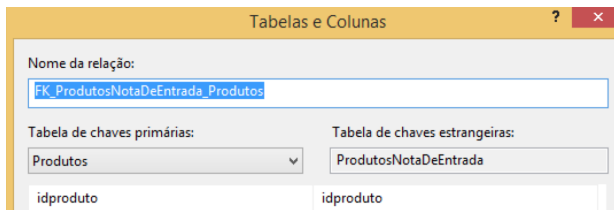



Fig. 6.59: Relacionamento entre as tabelas de Produtos de Notas de Entrada e Produtos

Com todas as tabelas e seus respectivos relacionamentos já criados, faz-se necessário disponibilizar essas tabelas no `DataSet`. Novamente, com o `DataSet` e a janela `Server Explorer` visíveis, arraste as tabelas de notas de entrada e produtos de notas de entrada para o `DataSet` (figura 6.60).

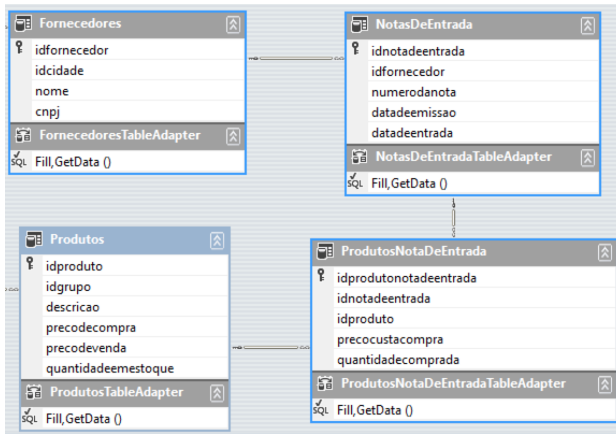



Fig. 6.60: Tabelas e relacionamentos relacionados às notas de entrada

Na janela de registro de notas de entrada, será feito uso dos recursos disponibilizados pelo `DataSet`. Desta maneira, na janela de `Data Sources`, configure o `DataTable` de notas de entrada, como mostra a figura a seguir:

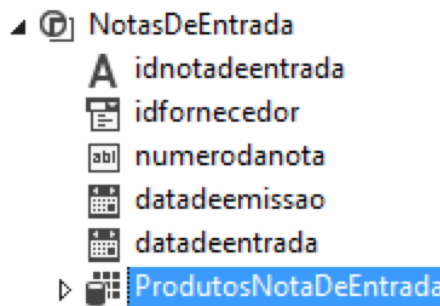



Fig. 6.61: Configuração do DataTable de notas de entrada para desenho e configuração do formulário

Arraste o `DataTable NotasDeEntrada` para o formulário, configure o `ComboBox` de `Fornecedores` para receber os dados do `DataTable Fornecedores` e os `DateTimePickers` para o formato `Short`. A configuração inicial para o formulário deverá ser semelhante ao apresentado na

figura seguinte.

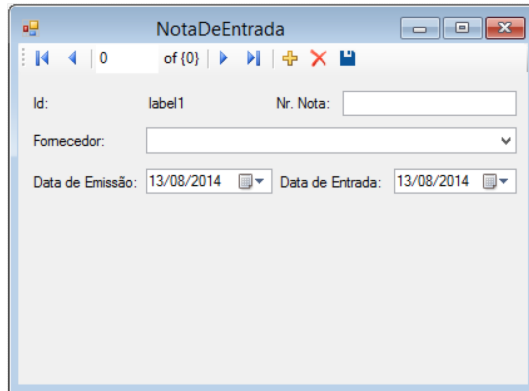



Fig. 6.62: Desenho inicial para a janela de Notas de Entrada, com dados da nota

A segunda parte da janela para registro das notas refere-se à informação dos produtos adquiridos na compra. Um primeiro passo é disponibilizar, por meio de um `DataGridView`, na área abaixo dos dados globais da nota, a relação de produtos existentes em cada nota, assim como a possibilidade de inserção, remoção e alteração desses produtos. Desta maneira, é preciso que os registros exibidos na nota apresentem o nome do produto, e não seu código.

Arraste na área livre do formulário o controle `GroupBox` (dentro da categoria `Containers` da `ToolBox`). Dentro dele, arraste um `BindingNavigator` (dentro da categoria `Data` da `ToolBox`) e, abaixo do `Navigator`, arraste a `DataTable` de produtos da nota de entrada, que está dentro da `DataTable` de notas de entrada. Na propriedade `BindingSource` do `Navigator`, selecione `produtosNotaDeEntradaBindingSource` – ou o nome que tenha o `DataSource` do `DataGridView` para produtos da nota de entrada, que foi criado quando o respectivo `DataTable` foi arrastado. Ao final, seu formulário deve estar semelhante ao apresentado na sequência pela figura:

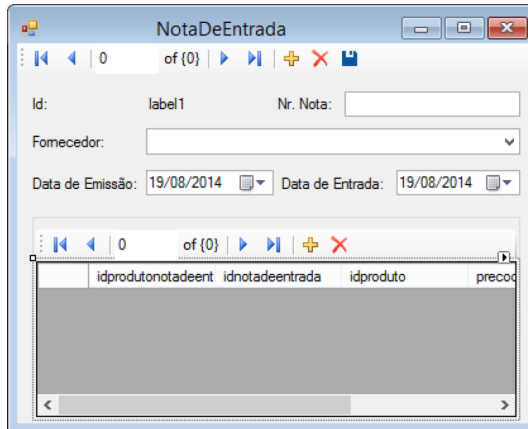



Fig. 6.63: Janela de notas de entrada com controles para manutenção dos produtos das notas

Como todos os campos foram inseridos e não será necessária a presença de todos no grid, na `SmartTag` do `DataGridView`, clique em `Edit Columns`. Deixe apenas as colunas: `idproduto`, `precocustocompra` e `quantidadecomprada`. Na propriedade `HeaderText` dessas colunas, deixe, respectivamente: `Descrição`, `Custo` e `Quantidade`. Na coluna `idproduto/Descrição`, na propriedade `ColumnType`, selecione `DataGridViewComboBoxColumn`. Este tipo de coluna tornará a coluna `idproduto/Descrição` um `ComboBox`, sendo possível selecionar o produto a ser inserido, diretamente da tabela de produtos. Finalize a alteração clicando no botão `OK` e o formulário agora deverá estar semelhante ao da figura a seguir:

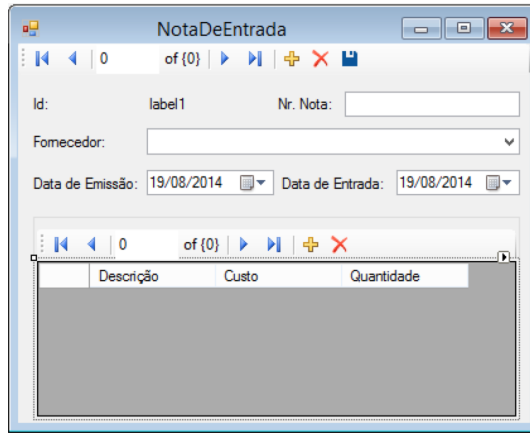



Fig. 6.64: Janela de notas de entrada com controles para manutenção dos produtos das notas configurados

Para a implementação da configuração do `ComboBox`, crie um novo método no `TableAdapter` de produtos para seleção dos dados, por ordem da descrição dos produtos. Na `ToolBox`, logo no início existe uma categoria com o nome do projeto e, dentro dela, existe o `TableAdapter` de produtos; arraste-o para a área de controles não visuais do formulário e tire o número 1 ao final de seu nome. Em seguida, implemente/altere o comportamento para o método que captura o evento `Load` do formulário, da maneira apresentada a seguir.

Resolução 6.13: método Load que popula os DataTables do formulário

```
private void NotaDeEntrada_Load(object sender, EventArgs e) {
    this.produtosTableAdapter.FillByDescricao(
    this.dSEstadosECidades.Produtos);

    this.produtosNotaDeEntradaTableAdapter.Fill(
        this.dSEstadosECidades.ProdutosNotaDeEntrada);
    this.fornecedoresTableAdapter.Fill(
        this.dSEstadosECidades.Fornecedores);
    this.notasDeEntradaTableAdapter.Fill(
```

```
    this.dSEstadosECidades.NotasDeEntrada);

    ((DataGridViewComboBoxColumn)dgvProdutos.Columns[0])
        .DataSource = this.dSEstadosECidades.Produtos;
    ((DataGridViewComboBoxColumn)dgvProdutos.Columns[0])
        .DisplayMember = this.dSEstadosECidades.Produtos.
        descricaoColumn.ColumnName;
    ((DataGridViewComboBoxColumn)dgvProdutos.Columns[0])
        .ValueMember = this.dSEstadosECidades.Produtos.
        idprodutoColumn.ColumnName;
}
```

Aqui, o primeiro `Fill` refere-se à população do `DataTable` que populará o `ComboBox` de produtos, configurado no final deste método também. Os demais `Fills` foram implementados automaticamente pelo Visual Studio, durante o processo de arrastar e soltar dos `DataTables`. As instruções finais referem-se à configuração do `ComboBox` para seleção de produtos no `DataGridView`. Elas são autoexplicativas, somadas ao conhecimento acumulado até este ponto do livro.

Execute a aplicação para testar o formulário de notas de entrada. Insira uma nota, informe os dados do corpo dela e grave-a. Clique no botão de adicionar item no `BindingNavigator` de produtos da nota e clique no `ComboBox` para produto. Sua tela deve estar semelhante à apresentada na figura:

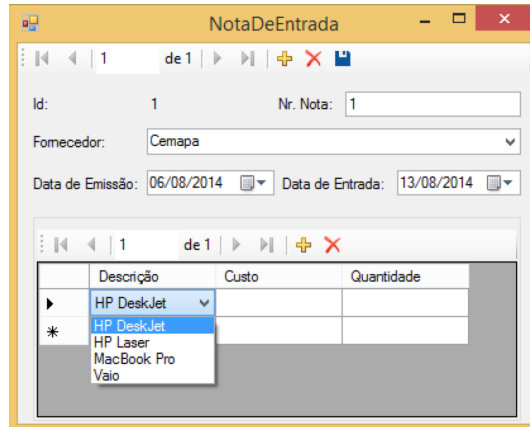



Fig. 6.65: Seleção de produto por meio de um ComboBox disponibilizado no DataGridView

Toda manutenção realizada no `DataGridView` só será persistida na base de dados com a gravação realizada pelo método que captura o evento `Click` do botão de gravar do `BindingNavigator`, do topo da janela. Caso os produtos não sejam gravados, verifique nas propriedades do `tableAdapterManager` se o `TableAdapter` para o `produtoNotasDeEntradaTableAdapter` está definido.

A janela que será proposta para o registro da venda terá um comportamento diferente do apresentado para a nota de entrada, pois ela contará com o fechamento da venda e realizará a verificação do estoque para o produto que se desejará vender. O primeiro passo é a criação das tabelas de notas de saída (figura 6.66) e de produtos das notas de saídas (figura 6.68), e seus respectivos relacionamentos (figura 6.67, 6.69 e 6.70).

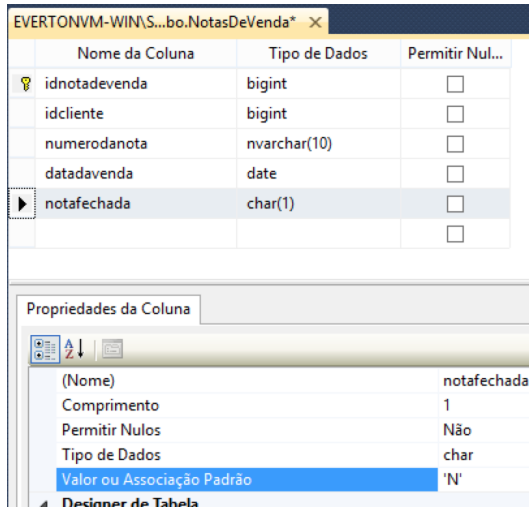



Fig. 6.66: Estrutura para a tabela de notas de saída

Verifique que a última coluna, `notafechada`, tem como valor padrão – que será atribuído aos novos registros – o caractere `'N'`, que foi atribuído à propriedade `Valor ou Associação Padrão`.

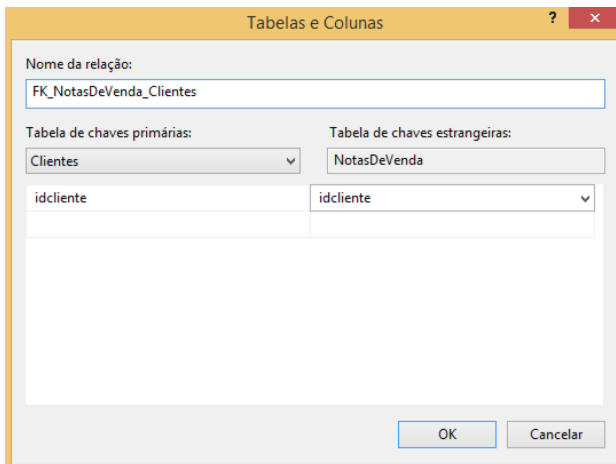



Fig. 6.67: Relacionamento entre as tabelas de NotaDeSaida e Clientes

Fig. 6.68: Estrutura da tabela Produtos para NotaDeSaida

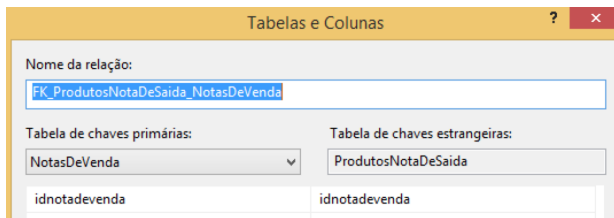



Fig. 6.69: Relacionamento entre as tabelas ProdutosNotaDeSaida e NotadeSaida

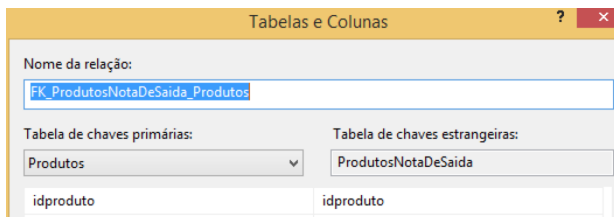



Fig. 6.70: Relacionamento entre as tabelas ProdutosNotaDeSaida e Produtos

Com todas as tabelas e seus respectivos relacionamentos já criados, faz-se necessário disponibilizar essas tabelas no `DataSet`. Novamente com o `DataSet` e a janela `Server Explorer` visíveis, arraste as tabelas de notas de saída e produtos de notas de saída para o `DataSet` (figura 6.71).

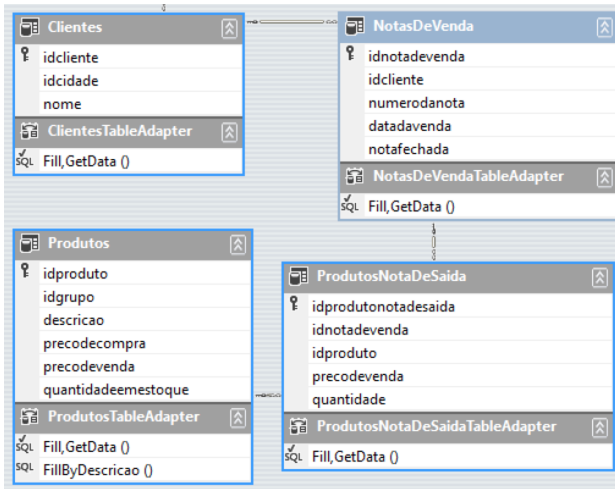



Fig. 6.71: Relacionamento entre as tabelas de produtos de notas de saída e de produtos

No formulário criado para o registro de notas de saída, arraste um `GroupBox` e, na propriedade `Text` dele, digite `Dados da Nota`. Configure o `DataTable` de notas de saída de acordo com o apresentado na figura 6.72.

Arraste o `DataTable` para dentro do `GroupBox`. Arraste a `DataTable` de clientes, da janela `Data Sources`, para o `ComboBox Cliente`. No `TableAdapter` de `Clientes`, no `DataSet` adicione um método `Fill()` que classifique os clientes por nome. Adapte no método que captura o evento `Load`, para que o método `Fill` do `adapter` de clientes seja o novo método criado. O formulário para as notas de saída, neste momento, deve estar semelhante ao da figura 6.73.

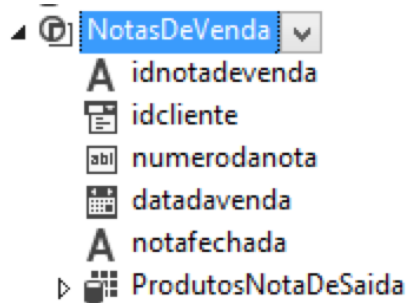



Fig. 6.72: Configuração dos controles para as DataColumns da nota de saída

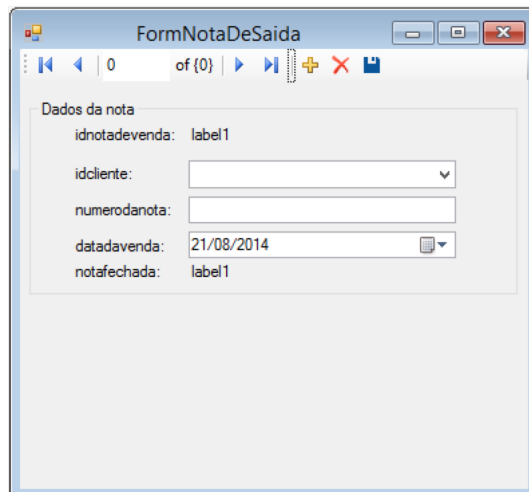
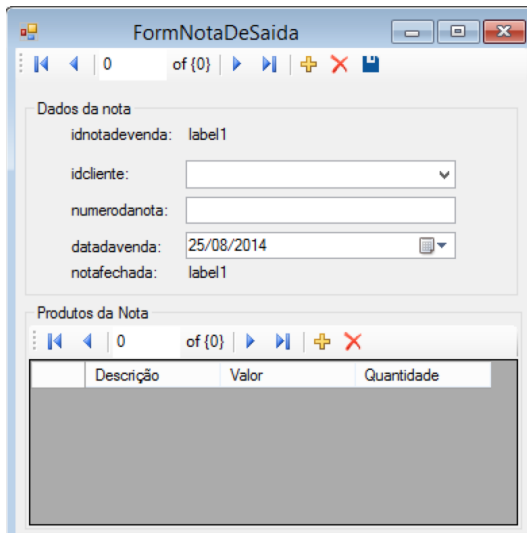



Fig. 6.73: Formulário de notas de saída com os dados da nota

Finalizando o desenho do formulário, arraste um novo `GroupBox`, agora abaixo do primeiro, e atribua à propriedade `Text` o valor `Produtos da Nota`. Arraste para dentro desse novo `GroupBox` um `BindingNavigator`, e configure sua propriedade `BindingSource` para o `Binding Source` de produtos da nota de saída.

Arraste para baixo do `Navigator` a `DataTable` de produtos da nota de saída, que está dentro da `DataTable` de notas de saída. Configure o

`DataGridView` de produtos, da mesma maneira que foi configurado o de notas de entrada, retirando as chaves estrangeiras e primárias, e modificando a coluna de produto para ser um `ComboBox`.

Arraste para a área de controles visuais o `TableAdapter`, que está na categoria de nome do projeto na `Toolbox`, e configure no `TableAdapterManager` a propriedade relativa a esse `TableAdapter`. Altere também o texto do cabeçalho das colunas.

Adapte o método para o evento `Load` para configurar esse `ComboBox`, tal qual foi feito para a nota de entrada. Seu formulário deve estar semelhante ao da figura a seguir:

Fig. 6.74: Formulário de notas de saída com os dados da nota e seus produtos

Algumas configurações agora serão realizadas para que cada nota possa ser fechada e ter seu estoque atualizado. Configure a propriedade `Enabled` dos dois `GroupBox` para `False`. Adicione um botão no `BindingNavigator` dos dados da nota e atribua uma imagem a ele, com a semântica de edição/alteração dos dados. Um segundo botão a ser inserido é um que se refira ao fechamento da nota, quando o estoque será, então,

atualizado. Um método para definição do estado dos botões precisa ser implementado. Veja o código na sequência.

Resolução 6.14: método Load que popula os DataTables do formulário

```
private void SetBindingNavigatorButtonsState() {
    bool podeFecharNota = false;
    if (notasDeVendaBindingSource.Current != null) {
        DataRowView drv = (DataRowView)
            notasDeVendaBindingSource.Current;
        DSEstadosECidades.NotasDeVendaRow nvRow =
            (DSEstadosECidades.NotasDeVendaRow) drv.Row;
        podeFecharNota = nvRow.notafechada.Equals("N");
    }
    bnbFirst.Enabled = !adding && !editing;
    bnbPrior.Enabled = bnbFirst.Enabled;
    bnbRecordNo.Enabled = bnbFirst.Enabled;
    bnbNext.Enabled = bnbFirst.Enabled;
    bnbLast.Enabled = bnbFirst.Enabled;

    bnbAdd.Enabled = bnbFirst.Enabled;
    bnbEdit.Enabled = bnbFirst.Enabled && podeFecharNota;
    bnbRemove.Enabled = !editing && podeFecharNota;
    bnbSave.Enabled = adding || editing;
    bnbFecharNota.Enabled = podeFecharNota &&
        (!adding && !editing);

    gbxDadosDaNota.Enabled = adding || editing;
    gbxProdutosDaNota.Enabled = editing;
}
```

O método `SetBindingNavigatorButtonsState()` tem, em sua primeira instrução, a definição de uma variável que terá, em base a algumas avaliações, o valor (verdadeiro ou falso), responsável por manter ativo ou desativo o botão relacionado ao processo de fechar uma nota.

O bloco `if()`, visto como segunda implementação do método, busca verificar se o registro corrente do `BindingSource` (`Current`) realmente

existe, pois a tabela pode estar vazia. Caso positivo, existe um registro corrente e recupera-se a linha atual. Essa recuperação se dá em duas linhas, sendo que a primeira recupera a `DataRowView`, e a segunda, a linha respectiva a tabela (`NotasDeVendaRow`, que é uma especialização de `DataRow`). Este processo poderia todo ser executado em uma única linha.

Sempre que um dado é exibido, como em um controle `DataGridView`, apenas uma versão de cada linha pode ser exibida e esta é um `DataRowView`. Um objeto dessa classe pode ter quatro estados diferentes: `Default`, `Original`, `Current` e `Proposed`. Após invocar o método `BeginEdit()` de uma `DataRow`, qualquer valor editado se tornará um valor `Proposed`. Até que o método `CancelEdit()` ou `EndEdit()` seja invocado, a linha (`row`) terá um valor `Original` e um `Proposed`. Caso seja chamado o método `CancelEdit()`, a versão proposta (`Proposed`) é descartada, e o valor é revertido para a versão `Original`. Se a invocação ocorrer para o método `EndEdit()`, a `DataRowView` não terá mais a versão `Proposed`, pois ela se tornará o valor corrente. Valores `Default` são disponíveis apenas para linhas que possuem definidos um valor padrão para as colunas. Para a implementação proposta, utilizou-se o `DataRowView` para obter o `DataRow`, referente ao `Current`.

Objetos das classes `DataRow` e `DataColumn` são componentes primários de uma `DataTable`. Esses objetos podem ser usados – assim como suas propriedades – para recuperar e avaliar, inserir, remover e atualizar valores de um `DataTable`. Na implementação, esse objeto `DataRow` foi utilizado para recuperar e avaliar o dado de um determinado campo.

O conjunto de instruções após a estrutura condicional `if` é responsável por definir o estado dos botões do `BindingNavigator`. Esse estado terá como base as variáveis `adding` e `editing`, responsáveis por informarem se o estado atual do processo é (ou não) de adição ou edição, pois o estado dos botões depende dessa situação. Esses campos foram definidos na declaração da classe, como mostro na sequência.

Resolução 6.15: definição de campos para a classe e do método construtor

```
public partial class FormNotaDeSaida : Form {
```

```
    private bool adding, editing;

    public FormNotaDeSaida() {
        InitializeComponent();
        adding = false;
        editing = false;
    }
//Resto da implementação omitido
}
```

O método **construtor** da classe inicializa as variáveis `adding` e `editing`, como `false`. Essa atribuição não seria necessária, pois `false` é o valor `default` para variáveis booleanas. Uma nova implementação foi necessária também no método que captura o evento `Load` do formulário. Veja o código a seguir.

Resolução 6.16: método que captura o evento Load do formulário

```
private void FormNotaDeSaida_Load(object sender, EventArgs e) {
// Implementação da Resolução 6.13

    if (notasDeVendaBindingSource.Current == null) {
        SetBindingNavigatorButtonsState();
    }
}
```

Na carga do formulário, caso não exista um registro corrente, o método `SetBindingNavigatorButtonsState()` é invocado. Esta verificação faz-se necessária para evitar uma chamada duplicada para o caso de existir um registro, pois quando existe, há a mudança de posição dos registros, disparando um evento para isso. Este evento é o `PositionChanged`, apresentado na sequência.

Resolução 6.17: método que captura o evento PositionChanged do BindingSource

```
private void notasDeVendaBindingSource_PositionChanged(
        object sender, EventArgs e) {
```

```
    DataRowView drv = (DataRowView)
        notasDeVendaBindingSource.Current;
    if (drv != null) {
        DataRow dr = drv.Row;
        if (drv != null && !(dr.RowState == DataRowState.
                Detached)) {
            SetBindingNavigatorButtonsState();
        }
    }
}
```

A condição para a chamada ao método `SetBindingNavigatorButtonsState()` é que exista uma `DataRowView` que não seja nula, e que o registro atual não seja um registro novo. Novamente, esta avaliação impede uma redundância a chamadas a esse método, quando for optado por inserir um novo registro (**Resolução 6.18**), que é visto como desconectado. Relembrando: as manutenções são todas realizadas em memória e, só ao final, quando optado por gravar (**Resolução 6.19**), são persistidas na base de dados.

Resolução 6.18: método que captura o evento Click do botão de Adicionar do BindingNavigator

```
private void bnbAdd_Click(object sender, EventArgs e) {
    adding = true;
    SetBindingNavigatorButtonsState();
}
```

No momento em que o usuário clicar no botão responsável por adicionar um novo registro, a variável `adding` recebe `true` para sinalizar esse estado. Então, o método `SetBindingNavigatorButtonState()` é invocado.

Resolução 6.19: método que captura o evento Click do botão de gravar do BindingNavigator

```
private void notasDeVendaBindingNavigatorSaveItem_Click(
        object sender, EventArgs e) {
    this.Validate();
```

```
    this.notasDeVendaBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(this.
        dSEstadosECidades);
    adding = false;
    editing = false;
    SetBindingNavigatorButtonsState();
}
```

Observe que, após a chamada ao método `UpdateAll()` do `tableAdapterManager`, há a redefinição das variáveis `adding` e `editing` para `false` e, em seguida, a invocação ao método `SetBindingNavigatorButtonsState()`.

É preciso agora implementar os métodos para o botão de alteração (**Resolução 6.20**) e o método de exclusão (**Resolução 6.21**). Veja que suas implementações são semelhantes a realizada para o método de adição de um novo registro.

Resolução 6.20: método que captura o evento Click do botão de alterar do BindingNavigator

```
private void bnbEdit_Click(object sender, EventArgs e) {
    editing = true;
    SetBindingNavigatorButtonsState();
}
```

Resolução 6.21: método que captura o evento Click do botão de remover do BindingNavigator

```
private void bnbRemove_Click(object sender, EventArgs e) {
    adding = false;
    editing = true;
    SetBindingNavigatorButtonsState();
}
```

Execute a aplicação e realize nela testes de inserção, alteração e remoção. Verifique o comportamento do estado dos botões.

Há um problema ainda com a implementação que precisa ser tratada, e fica como atividade para você implementar e resolver o problema. Para

identificar esse problema, crie um novo registro, grave-o e clique no botão `alterar`.

E se você não quiser alterar, mas sim descartar as mudanças? Pois é. Crie um novo botão para esta finalidade e, então, implemente o comportamento para ele. Anteriormente neste capítulo, foi apresentado o `DataRowView` e, nele, o método `CancelEdit()`. Realize essa atividade.

Agora, para finalizar o trabalho com o formulário de notas de venda, é preciso implementar o código para o botão `Fechar Nota`. Ao clicar nesse botão, o usuário terá a baixa do estoque do sistema e o fechamento da nota em si, não sendo permitido realizar alterações nela após essa operação. Para que isso possa ser realizado, alterações nas tabelas serão necessárias. Desta maneira, dois métodos precisarão ser criados. O primeiro será no `TableAdapter` de produtos. Relembrando, para criar esse método, clique com o botão direito do mouse sobre esse `TableAdapter`, no `DataSet`. Clique em `Add` e depois em `Query`, deixe selecionada a opção `Use SQL statements`, clique no botão `Next` e, em seguida, marque a opção `UPDATE`. No campo de código, informe a instrução SQL, visualizada na sequência.

Resolução 6.22: instrução SQL para o novo método do TableAdapter de Produtos

```
UPDATE Produtos SET
    quantidadeemestoque =
        @quantidadeemestoque - @quantidadevendida
    WHERE (idproduto = @idproduto)
}
```

Verifique que o `UPDATE` possui dois argumentos que serão utilizados: a quantidade vendida e o `Id` do produto que sofrerá a atualização.

Uma segunda instrução SQL precisa ser implementada, agora no `TableAdapter` de notas de venda. Esta alterará o valor do campo `notafechada` para `S`, o que impedirá alterações na nota. Veja o código mostrado a seguir.

Resolução 6.23: instrução SQL para o novo método do TableAdapter de Notas de Venda

```
UPDATE NotasDeVenda
    SET notafechada = 'S'
    WHERE (idnotadevenda = @idnotadevenda)
```

Execute a aplicação e realize nela testes referentes ao fechamento de nota. Valide o estoque e veja se ele está realmente sofrendo a baixa correta. Ainda, alguns pontos na aplicação ficaram sem implementação. Um deles é não realizar a baixa do estoque, se ele não possuir estoque suficiente. Outro ponto é implementar essa atualização do estoque também na nota de entrada.

6.12 Conclusão

Neste capítulo, recursos e práticas relacionados ao `DataSet` foram apresentados e trabalhados. Foi possível verificar que é possível o seu uso sem a necessidade de ter uma base de dados para populá-lo, mas que o uso mais comum é em sua forma DataSet Tipado.

Exemplos fazendo recursos de controles visuais, ligados ao DataSet e seus componentes, propiciaram uma visão de produtividade, fazendo uso dos recursos visuais oferecidos pelo Visual Studio. Ao final do exemplo, foi demonstrado o uso da criação de métodos nos `TableAdapter`s. O uso de DataSet Tipado é recomendado para pequenas aplicações, ou ainda para prototipação.

Nos capítulos seguintes, serão apresentados o *Language INtegrated Query* (LINQ) e o *Entity Framework* (EF), recursos fortemente recomendados para aplicações que tenham acessos à base de dados.

Capítulo 7

Conhecendo a Language INtegrated Query e o Windows Presentation Foundation

Language INtegrated Query (LINQ) é uma tecnologia desenvolvida pela Microsoft para fornecer suporte ao nível de linguagem – com recursos oferecidos pelo ambiente – e a um mecanismo de consulta de dados, para qualquer que seja o tipo deste conjunto de dados, que podem ser matrizes, coleções, documentos XML ou base de dados.

Como visto nos exemplos dos capítulos anteriores, as consultas a dados são expressas como `string`s, sem que haja uma verificação do compilador e sem oferecer um suporte por parte do IntelliSense, do Visual Studio. Em relação ao *Windows Presentation Foundation* (WPF), a Microsoft o aponta

como "a próxima geração" de sistemas de apresentação para construção de aplicações clientes, que, visualmente, faz um uso extremo da experiência do usuário.

Com o WPF, é possível fazer a construção de aplicações que funcionem tanto em ambientes desktop como em navegadores web. Este capítulo apresenta o LINQ e o WPF, trazendo conceitos iniciais e uma simples aplicação, que o permitirão ter uma noção sobre estes conteúdos.

7.1 Introdução ao Windows Presentation Foundation

O WPF é um mecanismo que possui um núcleo baseado em renderização vetorial, construído para tirar vantagens de hardwares gráficos mais modernos. Este framework estende um núcleo com um conjunto de recursos que inclui a XAML (*eXtensible Application Markup Language*), controles, ligação a dados, layout, gráficos 2D ou 3D, animações, estilos, templates, documentos, mídias e texto.

É possível utilizar o Visual Studio como ambiente para desenvolvimento de aplicações WPF, entretanto, sempre que possível, opte por ferramentas específicas, como o Blend.

7.2 Criando uma aplicação WPF

No Visual Studio, crie uma nova solução e, dentro dela, um projeto `WPF Application` (figura 7.1).

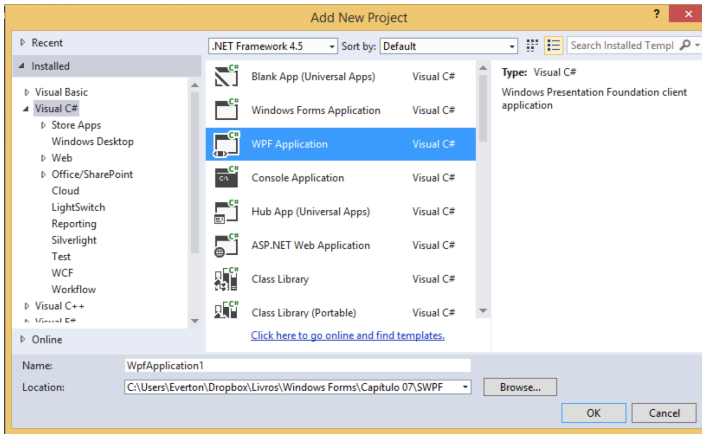



Fig. 7.1: Adicionando um projeto WPF Application a uma solução

Durante a criação do projeto, o Visual Studio faz uso dos templates para projetos WPF e, ao final da criação, exibe uma área de trabalho de acordo com o apresentado pela figura a seguir:

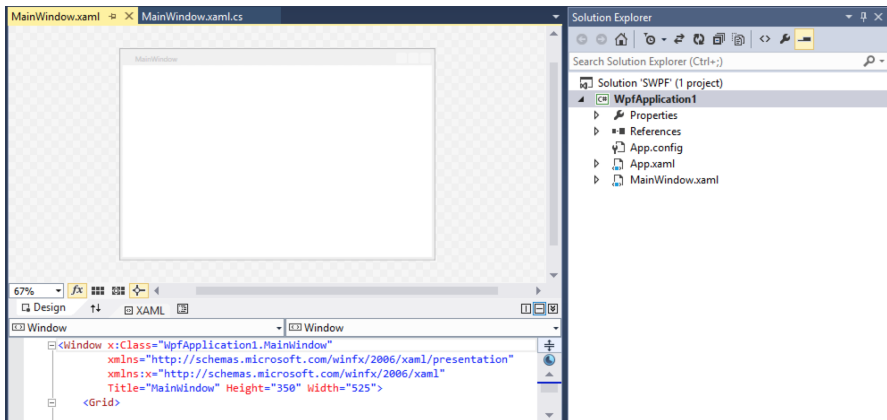



Fig. 7.2: Área disponibilizada pelo Visual Studio para implementação de um projeto WPF Application

A `Solution Explorer` apresenta três arquivos:

1) `App.config`, que possui o mesmo objetivo do arquivo de igual nome para projetos `Windows Forms Application`, que é o de possibilitar configurações para a aplicação/projeto.

2) `App.xaml`, que define a aplicação WPF e todos os recursos disponibilizados para ela. Também permite a especificação de qual interface com o usuário (UI – *User Interface*) será exibida automaticamente quando a aplicação for executada – no caso, `MainWindow.xaml` (**Resolução 7.1**).

3) `MainWindow.xaml`, que é o arquivo da janela principal da aplicação criada, e exibe o conteúdo criado em páginas (**Resolução 7.2**). A classe `Window` define as propriedades de uma janela (*window*) – como seu título, tamanho ou ícone – e gerencia eventos – como o fechamento (*closing*) ou a ocultação (*hiding*) da janela.

Resolução 7.1: código do arquivo MainWindow.xaml após a criação do projeto WPF Application

```
<Application x:Class="WpfApplication1.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Resolução 7.2: código da UI criada pelo template do projeto

```
<Window x:Class="WpfApplication1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
```

```
    </Grid>
</Window>
```

A XAML é uma linguagem declarativa que permite, dentre outras funcionalidades, inicializar objetos e configurar suas propriedades, utilizando uma estrutura de linguagem que apresenta relacionamentos hierárquicos. Por meio da XAML, é possível criar elementos visíveis para interfaces (UI) em uma tag XAML declarativa, separando em outro arquivo códigos que respondam a eventos. Isso permite a manipulação de objetos declarados em um XAML. Arquivos XAML são arquivos XML que possuem uma extensão `xaml`.

Em relação ao elemento `<Application.Resources>` do arquivo `App.xaml` (**Resolução 7.1**), `Application` expõe um escopo ao nível da aplicação para compartilhar recursos, nomeados como `Resources`. Por padrão, a propriedade `Resources` é inicializada com uma instância de `ResourceDictionary`. Essa instância é usada para obter e atribuir propriedades com escopo de aplicação, utilizando `Resources`.

Um `Grid` do elemento `<Grid>` (**Resolução 7.2**) é um painel, cujo layout organiza seus controles internos em uma estrutura tabular de linhas e colunas. Sua funcionalidade é similar a uma tabela HTML, entretanto, com maior flexibilidade de configurações. Uma célula pode conter vários controles e estes podem ocupar diversas células.

Para implementar um exemplo que possa introduzir tanto o WPF quanto o LINQ, será desenhada uma janela (figura 7.3 e **Resolução 7.3**) com uma lista de objetos de uma classe `Candidato`, que possui como propriedades `Nome` e `Idade`. A funcionalidade, que fará uso do LINQ, será implementada no evento que captura o evento `Click` de um botão.

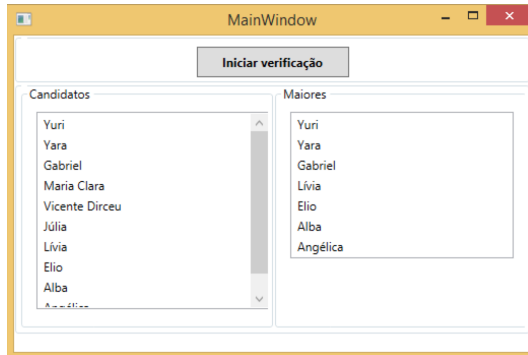



Fig. 7.3: Janela com botão, GroupBoxs e ListBoxs para aplicação do LINQ

A janela possui 4 `GroupBoxs`: um para o botão, outro para os dois `GroupBoxs` internos (`Candidatos` e `Maiores`). Existem outros containers que podem ser usados e trazer resultados visuais melhores, mas para cumprir o objetivo aqui proposto, o `GroupBox` atende bem.

O desenvolvimento da interface com o usuário pode ser realizado arrastando controles para a área de desenho (guia `ToolBox`), e pode ter suas propriedades e eventos configurados também de maneira visual (guia `Properties`), ou optar pela escrita do código, fazendo uso do IntelliSense do Visual Studio. Para facilitar a validação do código, inicialmente toda implementação terá seu código XAML apresentado. Na janela de "desenho", é possível alternar entre a visualização do código XAML e do resultado obtido para a codificação em questão. Também é possível dividir a janela com as duas visualizações, basta selecionar na base da área de desenho a opção desejada.

Resolução 7.3: implementação da janela da aplicação

```
<Window x:Class="WPFLINQIntroProject.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/
        xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/
        xaml"
    Title="MainWindow" Height="350" Width="525"
        HorizontalAlignment="Center" VerticalAlignment=
```

```
        "Center">
    <Grid>
<!-- Início da Parte 1 -->
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
<!-- Fim da Parte 1 -->

<!-- Início da Parte 2 -->
        <GroupBox Grid.Row="0">
            <Button Content="Iniciar verificação"
                HorizontalAlignment="Center" Margin="0,10,0,0"
                VerticalAlignment="Center" Width="150"
                Height="30" FontWeight="Bold" Click=
                "Button_Click"/>
        </GroupBox>
<!-- Fim da Parte 2 -->

<!-- Início da Parte 3 -->
        <GroupBox Grid.Row="1">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="250"/>
                    <ColumnDefinition Width="250"/>
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition Height="240"/>
                </Grid.RowDefinitions>
                <GroupBox Grid.Column="0" Header="Candidatos">
                    <ListBox Margin="10,10,0,13" Name=
                        "lbxCandidatos" HorizontalAlignment=
                        "Left" VerticalAlignment="Top"
                        Width="230" Height="Auto" />
                </GroupBox>
                <GroupBox Grid.Column="1" Header="Maiores">
                    <ListBox Margin="10,10,0,13" Name=
                        "lbxMaiores" HorizontalAlignment=
                        "Left" VerticalAlignment="Top"
```

```
                        Width="220" Height="Auto" />
                </GroupBox>
            </Grid>
        </GroupBox>
<!-- Fim da Parte 3 -->
    </Grid>
</Window>
```

No código, existem comentários delimitando conjuntos de instruções. Estes comentários e delimitações servirão para explicações das implementações. Na **Parte 1**, é trabalhada a definição das linhas para o `Grid` que está definido logo no início da implementação. A quantidade de linhas que ele possuirá está relacionada, neste caso, com a quantidade de linhas configuradas dentro do elemento `<Grid.RowsDefinitions>`.

A **Parte 2** define um `GroupBox`, que deverá ser alocado na linha 0, na primeira linha (atributo `Grid.Row="0"`). Dentro desse `GroupBox`, é definido um `Button`, com características definidas por meio de atributos para o elemento `Button`.

Já a **terceira parte** do código XAML define o `GroupBox` para a segunda linha do `Grid`. Nesse `GroupBox` é definido um `Grid`, com duas colunas e uma linha. Em cada coluna, é inserido um controle `ListBox`, que serão alimentados por meio de código.

Não faz parte do escopo do livro detalhar os elementos XAML e seus atributos. Entretanto, com o conhecimento adquirido até aqui, a leitura e interpretação semântica auxiliam na interpretação dos controles e suas configurações.

7.3 Introdução ao Language INtegrated Query

De acordo com a Microsoft, uma consulta é uma expressão que retorna dados de uma fonte de dados. Consultas são usualmente definidas em uma linguagem especializada para consultas. Diferentes linguagens e ferramentas têm sido desenvolvidas para os diversos tipos de dados existentes e que surgem, como por exemplo, a SQL para bancos de dados relacionais. Esta estratégia faz com os desenvolvedores precisem aprender novas linguagens para cada

tipo de fonte de dados (ou formatos de dados) que suas aplicações precisam suportar.

O objetivo do LINQ é simplificar essa situação, oferecendo um modelo consistente para trabalhar com diversos tipos de fontes e formatos de dados. Em uma consulta LINQ, o trabalho sempre ocorre sobre objetos. Desta maneira, o uso é o mesmo para consultar e transformar dados em documentos XML, bases de dados SQL, DataSets, coleções ou qualquer outro tipo de dado que ofereça um *provider* LINQ.

7.4 Preparando a aplicação para uma consulta LINQ

Para implementar o primeiro exemplo em LINQ, uma coleção de objetos da classe `Candidato`, apresentada na sequência, populará o `ListBox` criado na janela apresentada na figura 7.3.

Resolução 7.4: classe Candidato

```
namespace WPFLINQIntroProject {
    public class Candidato {
        public string Nome { get; set; }
        public int Idade { get; set; }
    }
}
```

Definido o tipo de dado que será trabalhado, é preciso criar uma coleção que seja populada com objetos do tipo definido – neste caso, `Candidato`. A seguir, apresento o método que possui essa responsabilidade.

Resolução 7.5: método que popula uma coleção de Candidatos

```
private void GenerateCandidatos() {
    candidatos.Add(new Candidato() {Nome = "Yuri",
        Idade = 25});
    candidatos.Add(new Candidato() {Nome = "Yara",
        Idade = 23});
```

```
    candidatos.Add(new Candidato() {Nome = "Gabriel",
        Idade = 20});
    candidatos.Add(new Candidato() {Nome = "Maria Clara",
        Idade = 7});
    candidatos.Add(new Candidato() {
        Nome = "Vicente Dirceu", Idade = 6});
    candidatos.Add(new Candidato() {Nome = "Júlia",
        Idade = 9});
    candidatos.Add(new Candidato() {Nome = "Lívia",
        Idade = 64});
    candidatos.Add(new Candidato() {Nome = "Elio",
        Idade = 74});
    candidatos.Add(new Candidato() {Nome = "Alba",
        Idade = 44});
    candidatos.Add(new Candidato() {Nome = "Angélica",
        Idade = 24});
}
```

A declaração e invocação do método `GenerateCandidatos()` pode ser verificada na sequência.

Resolução 7.6: declaração de coleção de candidatos e invocação de método para população de coleção e ListBox

```
public partial class MainWindow : Window {
    private List<Candidato> candidatos =
        new List<Candidato>();
    public MainWindow() {
        InitializeComponent();
        GenerateCandidatos();
        PopulateListBox();
    }

// Código omitido
}
```

Observe que, no construtor da classe que representa a janela, existe também a chamada ao método `PopulateListBox()` – apresentado a seguir

–, responsável por popular o `ListBox` com os `Candidatos` previamente populados na coleção `candidatos`.

Resolução 7.7: método que popula o ListBox de candidatos

```
private void PopulateListBox() {
    foreach (var candidato in this.candidatos) {
        lbxCandidatos.Items.Add(candidato.Nome);
    }
}
```

7.5 Implementando uma consulta LINQ

Toda operação de consulta LINQ consiste de três ações:

a) Obtenção da fonte de dados;

b) Criação da consulta;

c) Execução da consulta.

Essas três ações serão apresentadas na operação de consulta apresentada a seguir, que é disparada no evento `Click` do botão.

Resolução 7.8: método que popula o ListBox de candidatos

```
private void Button_Click(object sender, RoutedEventArgs e) {
// Início da consulta LINQ
    var maiores =
        from candidato in this.candidatos
        where candidato.Idade >= 18
        select candidato;
// Término da consulta LINQ

    foreach (var maior in maiores) {
        lbxMaiores.Items.Add(maior.Nome);
    }
}
```

Embora o exemplo proposto trabalhe em coleções, a sintaxe é a mesma para qualquer fonte de dados. No código, a primeira parte é a fonte de dados (neste caso, `candidatos`); já a segunda refere-se à criação da consulta, que é a variável `maiores` do tipo `IEnumerable<Candidato>`; e a última parte é a execução da consulta, que, neste exemplo, está no `foreach`.

A consulta (*query*) especifica que informação será recuperada da fonte de dados. Uma consulta pode também especificar como a informação deve ser classificada, agrupada e filtrada, antes de ser retornada. A consulta é armazenada em uma variável de consulta (*query variable*) e inicializada com a execução de uma expressão de consulta. Foi buscando tornar esse processo simples e fácil que a linguagem C# introduziu uma sintaxe para consultas.

Como já comentado, a variável de consulta *apenas* armazena em si as instruções para a realização de uma consulta. Sua real execução é adiada até que um resultado seja buscado. No exemplo apresentado, esse resultado veio por meio de uma iteração no conjunto retornado, pela instrução `foreach`.

Consultas que realizam funções de agregação sobre um intervalo de elementos precisam, primeiramente, iterar nesses elementos, o que causa uma **execução forçada**. Exemplos destas funções são `Count`, `Average` e `First`. Para esse caso, o tipo de dado da variável que recebe o resultado é de um valor simples, e não uma coleção `IEnumerable`. Para forçar uma execução imediata de qualquer consulta, guardando em cache seus resultados, é possível invocar o método `ToList<TSource>()` ou `ToArray<TSource>()`.

7.6 Conceitos básicos sobre consultas no contexto LINQ

Geralmente, a fonte de dados é organizada logicamente como uma sequência de elementos do mesmo tipo. Uma tabela de uma base de dados relacional contém uma sequência de linhas. De maneira similar, um `DataTable` contém uma sequência de objetos `DataRow`. Em um arquivo XML existe uma *sequência* de elementos XML, organizados hierarquicamente em uma árvore. Uma coleção disposta na memória local contém também uma sequência de objetos.

Do ponto de vista de uma aplicação, o tipo específico e a estrutura ori-

ginal da fonte de dados não são importante. A aplicação sempre vê a fonte de dados como uma coleção de `IEnumerable<T>` ou `IQueryable<T>`. Desta maneira, de uma maneira especializada, para o `LINQ to XML`, a fonte de dados é visível como um `IEnumerable<XElement>`; para o `LINQ to DataSet` é como `IEnumerable<DataRow>`; e para o `LINQ to SQL` é um `IEnumerable` ou `IQueryable` do tipo de dados dos objetos usados para representar os dados na tabela SQL.

Com a obtenção dessa sequência, uma consulta pode recuperar um subconjunto de elementos que permita produzir uma nova sequência, sem modificar os elementos individuais. Além disso, ela também pode classificar ou agrupar a sequência de várias maneiras. Um exemplo é apresentado na sequência.

Resolução 7.9: recuperando e classificando um conjunto de dados

```
var maiores =
    from candidato in this.candidatos
    where candidato.Idade >= 18
    orderby candidato.nome descending
    select candidato;
```

A leitura da instrução LINQ pode ser realizada da seguinte maneira: a instrução `select candidato` seleciona objetos retornados da instrução `from candidato in this.candidatos`. Por sua vez, os objetos retornados para `candidato` sofrem uma seleção, por meio de um critério (`where candidato.Idade >= 18`) e, em seguida, são classificados (`orderby candidato.nome descending`). Pelo fato de a exceção da leitura começar pela última linha, uma instrução LINQ assemelha-se às instruções SQL, e ela tem o recurso de autocomplemento, oferecido pelo Visual Studio.

Ainda com a obtenção da sequência, uma consulta também pode recuperar um valor único sobre uma fonte de dados, como: o número de elementos que correspondam a uma determinada condição; o elemento que possui o maior ou menor valor; o primeiro elemento que corresponda a uma condição; ou a soma de valores específicos em um determinado conjunto de elementos. Um exemplo para essa funcionalidade é apresentado a seguir.

Resolução 7.10: obtendo a quantidade de candidatos com base em um critério

```
var qtdeMaiores =
    (from candidato in this.candidatos
    where candidato.Idade >= 18
    orderby candidato.nome descending
    select candidato).Count();
```

Outra possibilidade para executar esse tipo de funcionalidade é obter a variável de consulta e, depois, invocar o método de agregação. Isso pode ser visto na sequência.

Resolução 7.11: obtendo a média da idade dos candidatos

```
var idadesCandidatos =
    from candidato in this.candidatos
    select candidato.Idade;
var mediaIdade = idadesCandidatos.Average();
```

7.7 Conceitos básicos sobre expressões de consultas

As expressões de consulta (*query expression*) vêm sendo utilizadas e explicadas conforme suas aplicações. Agora, é preciso detalhá-las um pouco mais. Desta maneira, uma expressão de consulta pode ser definida como uma consulta que é expressa em sintaxe específica para consultas. Ela é semelhante a qualquer outra expressão e pode ser usada em qualquer contexto em que uma expressão C# seja válida.

Essa expressão (*query expression*) é formada por um conjunto de cláusulas escritas em uma sintaxe declarativa, semelhante ao SQL. Cada cláusula dela pode, por si própria, conter uma ou mais expressões C#, e estas podem conter ou fazer parte de uma expressão de consulta.

Uma expressão de consulta precisa começar com uma cláusula `from`, e terminar com um `select` ou a cláusula `group`. Entre a primeira e última cláusula da expressão, é possível utilizar uma ou mais cláusulas opcionais,

como: `where`, `orderby`, `join`, `let` ou até mesmo cláusulas adicionais `from`. É possível também o uso da palavra-chave `into`, para permitir que o resultado de um `join` ou `group` sirva como fonte de dados para cláusulas adicionais de consulta, na mesma expressão.

7.8 Sintaxe de consultas e de métodos no LINQ

As consultas apresentadas até este momento fazem uso da sintaxe declarativa do LINQ. Entretanto, essa sintaxe de consulta deve ser traduzida em chamadas a métodos, para serem executadas na CLR (*Common Language Runtime*), o que ocorre no processo de compilação. Essas chamadas a métodos invocam os operadores padrões de consulta, os quais têm nomes como: `Where`, `Select`, `GroupBy`, `Join`, `Max` e `Average`.

Semanticamente, as sintaxes baseadas em consulta e em métodos são idênticas, porém, muitas pessoas veem a sintaxe de consultas como mais simples e fáceis de ler. Entretanto, é preciso ter ciência de que algumas consultas precisam necessariamente ser escritas com chamadas a métodos. Um exemplo é a necessidade de recuperar a quantidade de elementos (objetos/registros) que correspondam a uma determinada condição. A **Resolução 7.12** apresenta uma consulta usando a sintaxe de consultas, e a **Resolução 7.13**, a sintaxe de métodos.

Resolução 7.12: consulta com a sintaxe em consulta

```
var queryMaiores =
    from candidato in this.candidatos
    where candidato.Idade >= 18
    orderby candidato.Nome
    select candidato;

    foreach (var maior in queryMaiores) {
        lbxMaiores.Items.Add(maior.Nome);
    }
```

Resolução 7.13: consulta com a sintaxe em métodos

```
var queryMaiores =
    candidatos.Where(candidato => candidato.Idade >= 18).
    OrderBy(candidato => candidato.Nome);

    foreach (var maior in queryMaiores) {
        lbxMaiores.Items.Add(maior.Nome);
    }
```

O resultado (ou seja, os elementos) serão os mesmos em ambas implementações, inclusive o tipo de dados da variável `queryMaiores`, que é `IEnumerable<Candidato>`.

Na instrução que define a variável de consulta, a cláusula `where` é agora expressa como um método do objeto `candidatos`. Este não existe na classe `Candidato` e tampouco na interface `IEnumerable<T>`. Entretanto, na invocação do IntelliSense do Visual Studio, ao inserirmos o ponto (`.`) após a variável, é possível verificar que não apenas o método `Where()` é oferecido, mas diversos outros também, como `Select()`, `SelectMany()`, `Join()` e `Orderby()`. Existem métodos para todos os operadores padrões de consulta.

Apesar de termos a impressão que esses métodos foram redefinidos em `IEnumerable<T>`, não é isso que acontece. Eles, que são operadores básicos para consulta, são implementados como um tipo de método conhecido como `extension methods` (métodos de extensão). Esses métodos *estendem* um tipo existente, que podem ser invocados como se fossem métodos de instância do tipo do objeto. Os operadores básicos para consultas estendem `IEnumerable<T>` e é por isso que é possível utilizar `candidatos.Where(...)`.

Métodos de extensão (*extension methods*) habilitam a adição de métodos a um tipo (classe) já existente, sem a necessidade de estender uma classe para essa implementação. Eles podem ser vistos também como um tipo especial de método estático, porém, são chamados como se fossem métodos da instância em uma especialização.

7.9 Expressões Lambda (lambda expressions)

No exemplo apresentado anteriormente, verifique que a expressão condicional (`candidato.Idade >= 18`) é passada como um argumento `in-line`,

para o método `Where()`. Essa expressão `in-line` é chamada de expressão lambda, que é uma maneira conveniente para escrever código que poderia normalmente ser escrito de uma maneira mais trabalhosa, como por meio de métodos anônimos, de uma delegação genérica ou em uma expressão em árvores.

Em C#, o símbolo `=>` é o operador lambda, que é lido como "vai para". A variável `candidato`, à esquerda do operador, é a variável de entrada, que corresponde à variável `candidato` na expressão de consulta. O compilador pode inferir o tipo de `candidato` pelo fato de ele saber que `candidatos` é um tipo genérico `IEnumerable<T>`. O corpo da expressão lambda é exatamente como a expressão na sintaxe de consulta, ou em outras expressões C#; ele pode incluir chamadas a métodos e outras lógicas complexas. O "valor de retorno" é o resultado fornecido pela expressão.

No exemplo da **Resolução 7.13**, o método `OrderBy()` é invocado utilizando o operador ponto (`.`), após a chamada ao método `Where()`. O método `Where()` produz uma sequência selecionada (filtrada) e, então, o operador `Orderby` é aplicado a essa sequência, classificando-a. Devido ao retorno de consultas ser `IEnumerable`, é possível compor a consulta por meio da sintaxe de métodos, pelo encadeamento de chamadas aos métodos e uma "única" instrução. Este procedimento é o que o compilador realiza quando se escreve consultas por meio da sintaxe de consultas. Pelo fato de uma variável de consulta não armazenar os resultados, é possível modificá-la ou usá-la como base para uma nova consulta a qualquer momento, mesmo após ela ter sido executada.

7.10 Conclusão

As ferramentas, recursos e técnicas apresentados neste capítulo tiveram como objetivo uma breve introdução ao Language INtegrated Query (LINQ) e ao Windows Presentation Foundation (WPF), poderosos recursos que poderão trazer um ganho de qualidade e produtividade no acesso a dados, e uma nova possibilidade para geração de aplicações com recursos ricos para a interação com o usuário.

Com o exemplo apresentado e as explanações realizadas, verifica-se que o

uso do LINQ no processo de realização de consultas torna-se mais confiável, uma vez que faz uso de uma sintaxe que não utiliza de strings para a escrita de consultas. Também foi introduzido o conceito de *extension methods* (métodos de extensão) e *lambdas expressions* (expressões lambdas), que certamente serão usados amplamente em suas aplicações futuras.

Na apresentação do WPF, a janela criada, por mais que se pareça semelhante a uma aplicação `Windows Forms Application`, é possível verificar que a qualidade dos controles visuais é maior ao executarmos a aplicação. Na parte de implementação, foi possível conhecer o eXtensible Application Markup Language, e como implementar e codificar as interfaces com os usuários.

No próximo capítulo, será feito uso do Entity Framework (EF) e, nos exemplos que serão apresentados, os recursos deste capítulo serão úteis.

Capítulo 8

Apresentando o Entity Framework como ferramenta para Mapeamento Objeto Relacional

O Entity Framework (EF) é um framework para mapeamentos de objetos para um modelo relacional e de um modelo relacional para objetos (ORM – *Object Relational Mapping*). Por meio do EF, é possível trabalhar com dados relacionais fazendo uso de objetos da camada de negócio. Desta maneira, há uma eliminação de codificação de acesso a dados diretamente na aplicação, como por exemplo as instruções SQL.

Com essas características apresentadas, este capítulo tem por objetivo in-

troduzir o Entity Framework como ferramenta para persistência e interação com uma base de dados, tendo como apoio parte da aplicação já desenvolvida em capítulos anteriores. Não faz parte do escopo deste capítulo um aprofundamento no tema, que por si só já é tema único de diversos livros.

8.1 Começando com o Entity Framework

Para iniciar as atividades relacionadas ao desenvolvimento usando EF, o primeiro passo é obter os recursos necessários para utilizá-lo. A primeira ferramenta que precisamos disponibilizar no Visual Studio é o **EF Tools for Visual Studio**, que já vem instalada na versão 2013. A segunda ferramenta necessária é a **EF Runtime**, que deve ser disponibilizada nas referências (*References*) do projeto, como um Pacote/Assembly, em uma DLL. Para adicionar a referência ao EF Runtime, o meio mais simples é fazer uso do **NuGet**, que instalará no projeto selecionado o **EF NuGet Package**.

O **NuGet** é um gerenciador de pacotes para desenvolvimento na plataforma Microsoft, o que inclui o desenvolvimento de aplicações .NET. Essa ferramenta dispõe os pacotes a serem utilizados na aplicação em um servidor remoto, e também ferramentas clientes do NuGet, que permitem produzir e consumir pacotes NuGets.

8.2 Criando um projeto para aplicação do Entity Framework

O projeto inicial para aplicação do Entity Framework será uma aplicação Windows Presentation Foundation, com controles para entrada e visualização de dados por parte de usuário. É importante reforçar o conceito trabalhado anteriormente em relação a projetos separados, de acordo com as suas responsabilidades. Procure aplicar isso em suas aplicações.

Assim, crie uma nova solução e, dentro dela, um projeto WPF. Nesse projeto, crie uma pasta chamada `POCO` e, nela, a classe `Estado`. Veja o código a seguir.

Resolução 8.1: classe de negócio Estado (Versão 1)

```
public class Estado {
    public long Id { get; set; }
    public string UF { get; set; }
    public string Nome { get; set; }
}
```

Com a classe de modelo criada, já é possível começar a preparar o projeto para uso do Entity Framework. O primeiro passo é a criação do contexto, que representará a conexão com a base de dados. Como a criação desse contexto faz uso da classe `System.Data.Entity.DbContext` e dispõe uma propriedade do tipo `DbSet<TEntity>`, é preciso adicionar a referência ao NuGet Package do Entity Framework.

8.3 Habilitando a API do Entity Framework (NuGet Package to Entity) para o projeto

Para que o projeto em desenvolvimento possa fazer uso do Entity Framework, clique com o botão direito do mouse sobre o nome do projeto e, então, na opção `Manage NuGet Packages` (figura 8.1).

Fig. 8.1: Acessando a gerência de pacotes NuGet

Na janela exibida, é preciso selecionar a categoria de pacotes que serão exibidos/pesquisados. Clique do lado esquerdo na opção `on-line` e, na caixa de busca do lado direito, digite **Entity Framework**. Como resultado, sua janela estará semelhante à apresentada na figura a seguir:

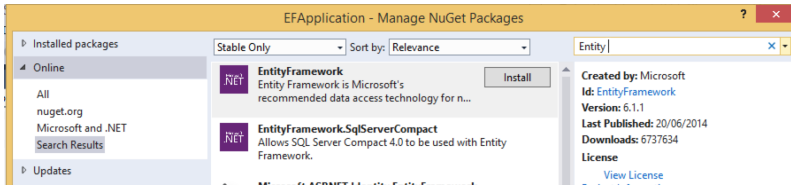



Fig. 8.2: Buscando o pacote NuGet para instalação

O NuGet Package para o Entity Framework deverá aparecer como primeiro item, de acordo com a figura 8.2. Assim, clique no botão `Install`, que aparece ao lado direito do nome do pacote. Antes do processo de instalação começar, é solicitada a leitura do termo da licença de uso e confirmação do aceite (figura 8.3).

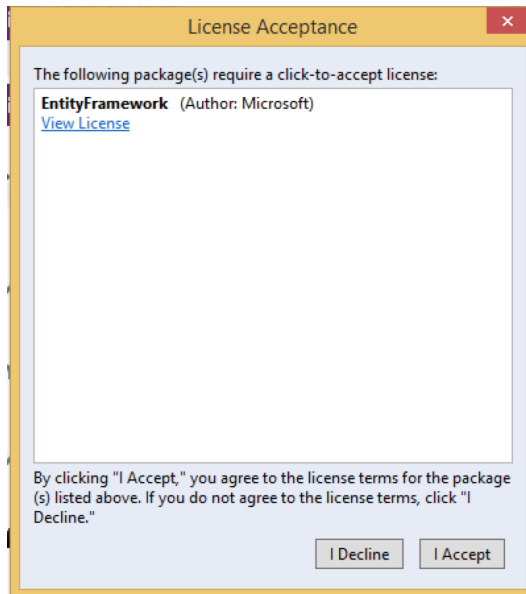



Fig. 8.3: Leitura e concordância com o termo de licença de uso do NuGet Package para o EF

Após a instalação, note a marca ao lado do nome do pacote, que indica

que ele já está instalado (figura 8.4).

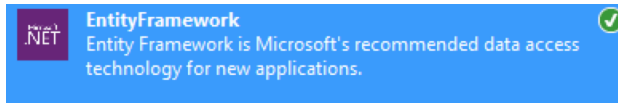



Fig. 8.4: Indicação da instalação bem sucedida do NuGet Package para o EF

Após fechar a janela de manutenção dos pacotes NuGets, verifique a adição às referências do projeto dos Assemblies do Entity Framework (figura 8.5).

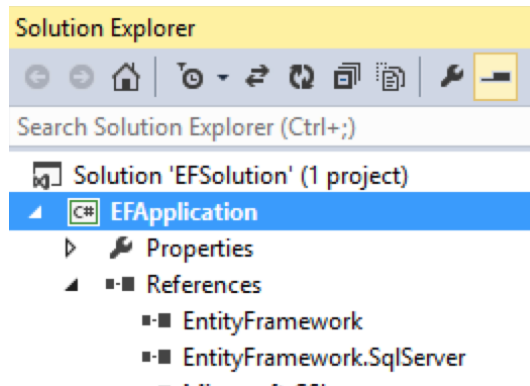



Fig. 8.5: Assemblies do EF adicionados às referências do projeto

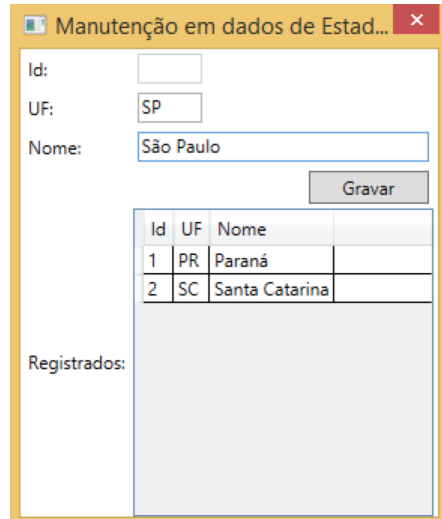
Quando um pacote é instalado, o NuGet copia arquivos para a solução e automaticamente realiza todas as alterações que são necessárias, como a adição de referências e mudanças no `app.config`. Se decidir remover a biblioteca, o NuGet remove os arquivos e reverte todas as alterações realizadas no projeto por ele.

8.4 Criando o contexto com a base de dados

Para que aplicações .NET possam se beneficiar com o Entity Framework, é preciso que o framework acesse a base de dados por meio de um contexto,

que representará uma sessão de interação, seja para consulta ou atualização da aplicação com a base de dados.

Para o EF, um contexto é uma classe que estenda `System.Data.Entity.DbContext`. Desta maneira, crie em seu projeto uma pasta chamada `Contexts` e, dentro dela, uma classe `EFContext.cs`, que deverá ter essa extensão implementada. Veja o código na sequência.

Resolução 8.2: classe que representará o contexto com a base de dados (Versão 1)

```
using System.Data.Entity;
using EFApplication.POCO;

namespace EFApplication.Contexts {
    public class EFContext : DbContext {
        public DbSet<Estado> Estados { get; set; }
    }
}
```

No código, a primeira implementação diz respeito à importação do `namespace` relativo ao EF, o `System.Data.Entity;`. Na declaração da classe, a extensão de `DbContext` é realizada (`:DbContext`) e, por fim, uma propriedade do tipo `DbSet<Estado>`, chamada `Estados`, é criada.

Propriedades da classe `DbSet` representam entidades (`Entity`) que são usadas para as operações de criação, leitura, atualização e remoção de objetos (registros da tabela). Desta maneira, se na base de dados há uma tabela que será manipulada por sua aplicação por meio do EF, é importante que, na definição do contexto, haja uma propriedade que a represente.

8.5 Lendo e escrevendo dados na base de dados por meio do EF

Para exemplificar a inserção de um objeto à base de dados e recuperar todos os já persistidos, será implementado um exemplo, fazendo uso de WPF com EF e das classes já implementadas neste capítulo. As figuras 8.6 e 8.7 apresentam a interface que deverá ser implementada para a interação com o usuário.

Fig. 8.6: Janela de interação com o usuário para registro de Estados – Inserindo novo estado

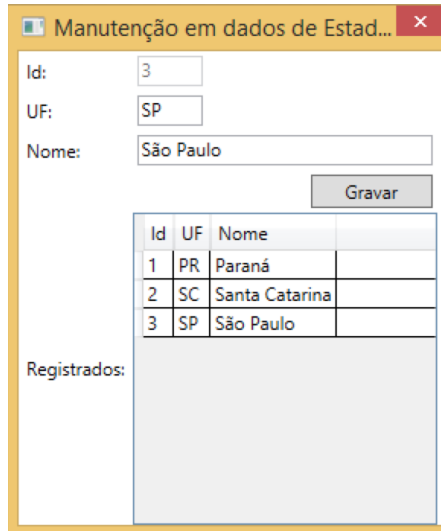



Fig. 8.7: Janela de interação com o usuário para registro de Estados – Alterando um estado já gravado

As figuras 8.6 e 8.7 possuem três áreas. A **primeira** é a de entrada de dados, para `Id`, `UF` e `Nome do Estado`. Em relação ao `Id`, ele não tem seu valor informado pelo usuário, mas sim pela base de dados e, após a inserção, este é atribuído ao objeto que representará o estado inserido.

Já a **segunda** área é referente ao botão `Gravar`, que fará uso dos dados informados nos `TextBoxs` para instanciar um objeto de `Estado` e, então, enviá-lo para persistência por meio do EF. A **terceira** e última parte refere-se à apresentação de todos os estados já persistidos, por meio de um `DataGrid`. A seguir, trago a implementação desta janela.

Resolução 8.3: XAML da janela de manutenção nos dados de estado

```
<Window x:Class="EFApplication.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Manutenção em dados de Estados" Height="350"
```

```
        Width="293.656" WindowStartupLocation=
        "CenterScreen" ResizeMode="NoResize">
    <Grid>
<!-- Definição e configuração de linhas e colunas para o Grid
     (painel da janela) -->
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="28" />
            <RowDefinition Height="*"  />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="200" />
        </Grid.ColumnDefinitions>
<!-- Registro e configuração dos controles que comporão o
     formuário -->
        <Label Grid.Row="0" Grid.Column="0" Content="Id:"/>
        <Label Grid.Row="1" Grid.Column="0" Content="UF:"/>
        <Label Grid.Row="2" Grid.Column="0" Content="Nome:"/>
        <Label Grid.Row="4" Grid.Column="0"
            Content="Registrados:" VerticalAlignment=
            "Center"/>
        <TextBox x:Name="txtID" Grid.Column="1" Grid.Row="0"
            Margin="3,3,154,3" IsEnabled="False" />
        <TextBox x:Name="txtUF" Grid.Column="1" Grid.Row="1"
            Margin="3,3,154,3" />
        <TextBox x:Name="txtNome" Grid.Column="1" Grid.Row="2"
            Margin="3" />
        <Button Grid.Column="1" Grid.Row="3"
            HorizontalAlignment="Right" MinWidth="80"
            Margin="3" Content="Gravar" />
<!-- Controle responsável por exibir todos os estados já
     persistidos -->
        <DataGrid x:Name="dgEstados" Grid.Row="4"
            Grid.Column="1" AutoGenerateColumns="True"
            IsReadOnly="True" >
        </DataGrid>
```

```
    </Grid>
</Window>
```

O controle DataGrid do WPF permite que um conjunto de dados (de uma base de dados ou coleção) seja exibido de maneira tabular, semelhante ao DataGridView do Windows Forms, porém, com maiores recursos. As funções básicas oferecidas por este controle são: geração automática (ou manual) de colunas, seleção de linhas e colunas, agrupamento de dados, classificação por colunas, redimensionamento e reordenamento das colunas, congelamento (fixação) de colunas, dentre outras.

Para que o DataGrid possa ser populado com os estados já persistidos na base de dados, é preciso a implementação de um método para isso. Na sequência, mostro um método que recupera, por meio do EF, esses estados.

Resolução 8.4: método que retorna uma lista

```
private IList<Estado> GetEstados() {
    using (var context = new EFContext()) {
        return context.Estados.ToList<Estado>();
    }
}
```

Para retornar todos os estados persistidos, por meio do EF, é preciso ter disponível o contexto de acesso a base de dados, que é fornecido pela classe EFContext (**Resolução 8.2**). Por meio do contexto, obtém-se a propriedade Estados, que é uma IQueriable<Entity> e, então, a consulta é realizada por meio da invocação ao método ToList(), que transforma o resultado em uma lista genérica.

Na sequência, é preciso criar um método que seja responsável por atualizar o DataGrid, que sofrerá atualização quando a janela for instanciada e a cada inserção que ocorrer. A preferência em criar um método específico para isso está ligada aos princípios de responsabilidades dos métodos.

O método GetEstados() tem a responsabilidade de fornecer os dados, e não de atualizar o DataGrid. A seguir, apresento o método responsável por atualizar o DataGrid.

Resolução 8.5: método que atualiza o DataGrid

```
private void RefreshDataGrid() {
    dgEstados.ItemsSource = GetEstados();
}
```

Como pode ser verificado no código do método `RefreshDataGrid()`, a propriedade do `DataGrid` responsável por manter os dados a serem exibidos é a `ItemsSource`, que recebe o retorno do método `GetEstados()`. Essa propriedade também pode ser configurada no código XAML, quando existe um `Binding` (ligação) do controle com uma fonte de dados.

A primeira invocação a esse método deve ocorrer na instanciação da janela, ou seja, no construtor da classe que a representa. Veja a seguir.

Resolução 8.6: método Construtor da classe que representa a Janela

```
public MainWindow() {
    InitializeComponent();
    RefreshDataGrid();
}
```

Ao executar a aplicação pela primeira vez, nada será exibido no `DataGrid`, pois nada foi inserido. Na realidade, nem a base de dados e a tabela foram construídas. Para a inserção dos dados informados na janela, é preciso um método específico para isso, que é apresentado na sequência.

Resolução 8.7: método que realiza a inserção de um estado na base de dados

```
private Estado SaveEstado(Estado estado) {
    using (var context = new EFContext()) {
        context.Estados.Add(estado);
        context.SaveChanges();
    }
    return estado;
}
```

Verifique que o procedimento de obtenção do contexto para acesso à base de dados pelo EF é o mesmo apresentado na **Resolução 8.4**, tendo como alteração a invocação ao método `Add()` da propriedade `Estados`, enviando o objeto recebido como argumento.

Neste exemplo, foi realizada apenas uma inserção de objetos gerenciáveis pelo contexto. Entretanto, é possível que ocorram várias, sendo que, ao final dessas operações, é preciso invocar o método `SaveChanges()` oferecido pela `DbContext`.

O retorno do mesmo objeto recebido ao método chamador permite que atualizações realizadas durante o processo de atualização sejam disponibilizadas para a aplicação, como é o caso do valor para a propriedade `Id`, que é gerada quando o objeto é persistido.

Para que uma propriedade de uma classe seja identificada como chave primária para a tabela, basta que seja nomeada de `Id`. Caso precise nomear a propriedade que representa a chave primária com outro nome, será preciso fazer uso de `Data Annotations`.

Para que os dados informados pelo usuário possam ser persistidos por meio do método `SaveEstado()`, implemente o código que trago a seguir, no método que captura o evento `Click` do botão `Gravar`.

Resolução 8.8: método que captura o evento Click do botão Gravar

```
private void btnGravar_Click(object sender,
        RoutedEventArgs e) {
    var estado = SaveEstado(new Estado(){
        UF = txtUF.Text,
        Nome = txtNome.Text
    });
    txtID.Text = estado.Id.ToString();
    RefreshDataGrid();
}
```

Com essa implementação, execute a aplicação, insira um estado e veja que, após a execução, o `DataGrid` o exibe. Feche a aplicação e execute novamente, o estado deve aparecer.

8.6 Identificando as ações do Entity Framework

Talvez, a primeira dúvida na execução do projeto seja saber onde foi criada a base de dados. Por padrão (*default*), o Entity Framework cria-a automaticamente, pois nenhuma informação sobre ela foi registrada na aplicação criada.

A instância do SQL Server utilizada depende dos recursos que você instalou em sua máquina: ou SQL Server completo, ou o Express Edition, ou ainda o Local Db. A base de dados é criada, por padrão, com o nome qualificado. Para o exemplo, o nome usado foi `EFApplication.Contexts.EFContext`, ou seja, **NomeDoProjeto.Namespace.Classe**. Para visualizar e manipular a base de dados no Visual Studio, na janela `Server Explorer` clique com o botão direito em `Data Connections` e, em seguida, em `Add Connection` (figura 8.8).

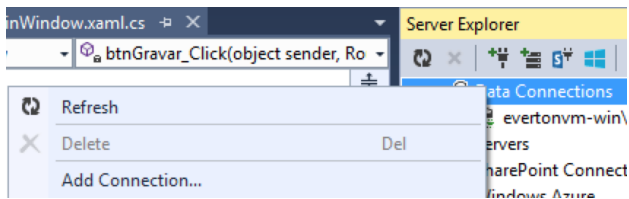



Fig. 8.8: Adicionando uma nova conexão com o SQL Server Express

Na janela que se abre, informe o endereço do servidor e a instância do SQL Server Express, e depois selecione a base de dados, que deve aparecer na lista das disponíveis (figura 8.9).

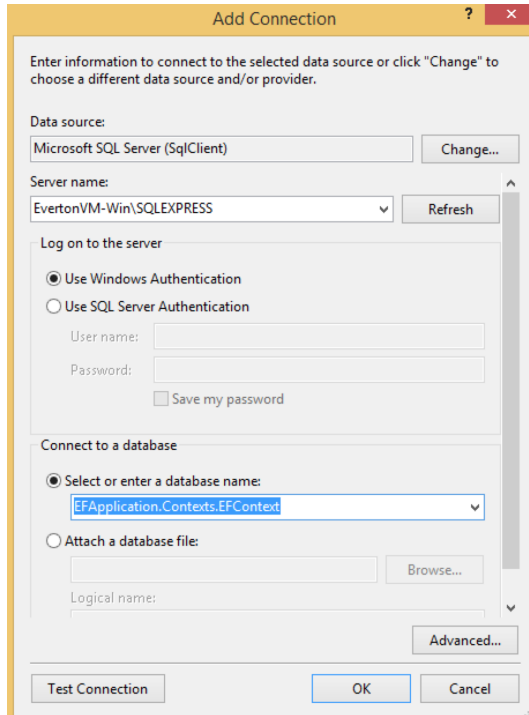



Fig. 8.9: Configurando a conexão com a base criada pelo Entity Framework

Após ter configurado e testado a conexão, confirme sua criação clicando no botão `OK`, e verifique que ela aparece no `Server Explorer` (figura 8.10).

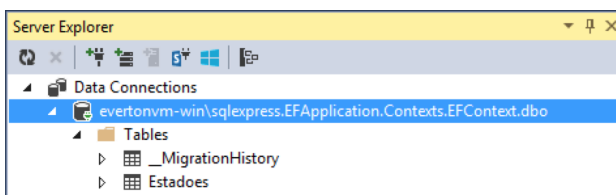



Fig. 8.10: Visualização da base de dados e tabelas criadas pelo Entity Framework

Verifique na lista de tabelas a existência de duas: `_MigrationHistory`

e `Estadoes`. A primeira faz parte de um recurso, que não será trabalhado neste livro, chamado `Code First Migration`, e a segunda representa os estados persistidos. Note que o nome da tabela sofreu um processo de **pluralização**, porém, seguindo o idioma inglês. Essa pluralização pode ser desabilitada.

O **Entity Framework Code First Migration** é um recurso oferecido pelo EF para auxiliar nas atualizações realizadas em uma base de dados. Normalmente (e principalmente) quando se desenvolve uma aplicação, o modelo de negócio sofre alterações, e estas refletem na base de dados e na estrutura das tabelas quando utilizamos o EF.

Realizar este controle de maneira manual é um trabalho árduo. Normalmente, o desenvolvedor recorre a ferramentas de terceiros para essa atividade. O EF 6 trouxe o Entity Framework Code First Migration para subsidiar essa atividade. Apesar de não ser parte do escopo deste livro, a habilitação desse recurso ocorre por padrão, mas também pode ser desabilitada. Recomenda-se a investigação e uso efetivo em seus projetos do Migrations.

Code First é uma das **estratégias** disponibilizadas pelo Entity Framework para o mapeamento entre objetos e tabelas (ORM) e é a utilizada neste livro, na qual se define o modelo de negócio. Por meio desse modelo, o EF cria e mantém atualizada a base de dados e permite a interação com os dados, para assim obter, fornecer, atualizar e remover objetos.

As demais estratégias Model First e Database First não fazem parte do escopo deste livro, mas é interessante uma investigação sobre elas, principalmente se for realizado um desenvolvimento sobre uma base de dados existente.

8.7 A inicialização da base de dados

O fluxo seguido pelo EF no processo de inicialização da base de dados, no exemplo apresentado, foi o de verificar se o construtor da classe que estende `DbContext` (`EFContext`, no caso) recebe uma `Connection String`. Se essa condição fosse verdadeira, a configuração recebida conteria as definições para criar a base de dados, ou abri-la caso já existisse.

Como no exemplo não foi passada uma `Connection String`, o EF

passa à segunda verificação, que é a de o construtor receber uma `string` que contém o nome da base de dados que será criada, ou aberta caso ela exista. O que ocorreu no exemplo trabalhado é que nem uma `Connection String` nem uma `string` com o nome para a base de dados foi informada, o que resultou na criação da base de dados com o nome qualificado.

Adapte a implementação da classe `EFContext`, de acordo com o apresentado a seguir.

Resolução 8.9: classe que representará o contexto com a base de dados (Versão 2)

```
using System.Data.Entity;
using EFApplication.POCO;

namespace EFApplication.Contexts {
    public class EFContext : DbContext {
        public EFContext() : base("EF_Intro") {}
        public DbSet<Estado> Estados { get; set; }
    }
}
```

Execute a aplicação. Note que os estados inseridos anteriormente não existem mais, pois uma nova base de dados foi criada com a mudança realizada na classe `EFContext`. No `Server Explorer`, adicione essa nova base de dados, da mesma maneira que inseriu a anterior. Para testar com uma `Connection String`, crie-a no `app.config` e informe no construtor o nome dado a ela.

8.8 Implementando associações/ relacionamentos

Para implementar no EF associações entre classes que se mapeiam em relacionamentos de tabelas, a princípio segue-se o conceito básico de Orientação a Objetos, trazendo apenas algumas características que possibilitem uma melhor navegação (navegabilidade) entre os objetos da associação. A seguir, apresento a classe `Cidade`, que se associa com `Estado`.

Resolução 8.10: implementação da classe Cidade

```
public class Cidade {
    public long Id { get; set; }
    public string Nome { get; set; }

    public long EstadoId { get; set; }
    public virtual Estado Estado { get; set; }
}
```

Verifique no código da classe que a referência para um objeto `Estado` ocorre de duas maneiras. Na primeira, onde a propriedade `EstadoId` possui o tipo `long`, é armazenada a chave estrangeira de estado, ou seja, a chave primária da tabela `Estado`. A segunda maneira especifica uma propriedade do tipo `Estado`, precedida da palavra-chave `virtual`. Métodos e propriedades definidos como virtuais permitem a sobrescrita (*override*) por subclasses em C#, que é um processo que ocorre quando usamos frameworks, como o EF. Assim, ao implementarmos uma subclasse de `Cidade`, o EF pode aplicar o processo de carga tardia (`lazy loading`).

Quando um modelo de negócio possui classes associadas, o framework de ORM (EF) precisa trazer, além do objeto recuperado, o(s) objeto(s) associado(s) a ele (registro). Este processo precisa ser bem pensado, para não causar uma sobrecarga desnecessária na aplicação.

Pensando na associação proposta de `Estado` e `Cidade`, é preciso avaliar se, ao recuperar um estado, todas as cidades pertencentes a ele também precisam ser carregadas no mesmo momento (*eager*), ou apenas quando forem necessárias para a aplicação (*lazy*).

O Entity Framework oferece suporte para três estratégias de recuperação de dados de uma associação/relacionamento: `eager loading`, `lazy loading` e `explicit loading`. Este último pode ser utilizado quando temos o `lazy loading` desativado por `default`, mas em algum momento se deseje que certa associação tenha uma carga tardia.

Para garantir uma associação com navegabilidade bidirecional, é preciso alterar a classe `Estado`, conforme mostrado na sequência.

Resolução 8.11: classe de negócio Estado (Versão 2)

```
public class Estado {
    public long Id { get; set; }
    public string UF { get; set; }
    public string Nome { get; set; }

    public virtual List<Cidade> Cidades { get; set; }
}
```

Mesmo não tendo implementado uma interface com o usuário (Janela) para o registro de `Cidades`, execute sua aplicação. Durante a execução, uma janela de erro semelhante à da figura 8.11 será exibida.

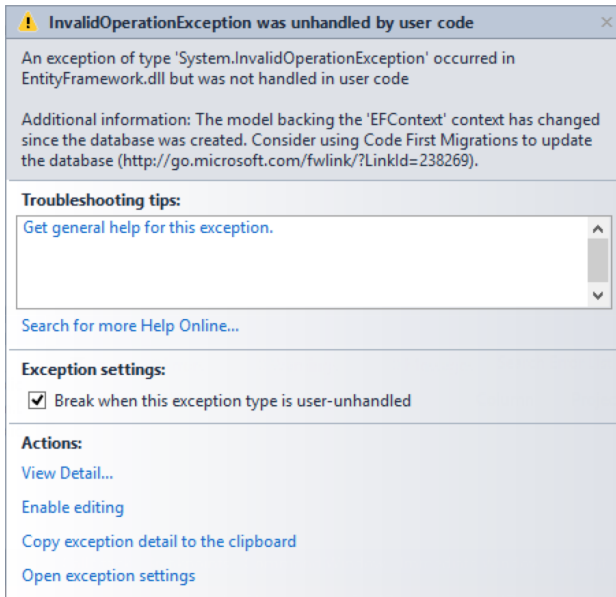



Fig. 8.11: Erro de validação de modelo encontrado pelo Entity Framework

8.9 Estratégias para inicialização da base de dados

Vamos recapitular: na primeira etapa da aplicação de exemplo, apenas uma classe foi implementada e a execução ocorreu sem problema algum. A base de dados foi criada, registros foram inseridos, a aplicação foi fechada e iniciada outras vezes, e tudo ocorreu perfeitamente. Já na segunda etapa, foi aplicado o conceito de associação, por meio da implementação da classe `Cidade` e das propriedades relacionadas. Ao executar, a exceção apresentada na figura 8.11 ocorreu. Essa exceção (`System.InvalidOperationException`) ocorre pelo fato de que, no momento em que o EF busca o contexto (`EFContext`) com a base de dados, uma validação do modelo de negócio (classes) com as tabelas (base de dados) é realizada e, como mudanças foram feitas, elas não casam – conforme informação adicional na janela da exceção: *Additional information: The model backing the 'EFContext' context has changed since the database was created. Consider using Code First Migrations to update the database* (http://go.microsoft.com/fwlink/?LinkId=238269) .

O que se espera é que, quando alterações forem realizadas no modelo de negócio (classes), estas atualizações sejam propagadas para a base de dados. Para isso, é preciso pensar em estratégias para inicialização do contexto com a base de dados, ou seja, o que o EF deve realizar quando for realizar a conexão.

Existem quatro possíveis estratégias:

- `CreateDatabaseIfNotExists`: é a estratégia padrão para o EF. Como pode ser compreendido pelo nome, a base de dados será criada caso ela não exista, ao realizar a conexão. Entretanto, quando modificações são feitas no modelo de negócio, ao executar a aplicação, há a ocorrência de exceções, que foi o que aconteceu no exemplo anterior.

- `DropCreateDatabaseIfModelChanges`: fazendo uso dessa estratégia, quando ocorrer alterações no modelo de negócio, a base de dados é removida e uma nova é criada, caso ela exista. Essa deveria ser a estratégia para o exemplo apresentado. Entretanto, se sua aplicação já possuir dados que foram inseridos para testes, eles vão ser perdidos. O EF oferece recursos para popular uma base de dados com dados para testes.

- `DropCreateDatabaseAlways`: novamente, a leitura do nome dessa estratégia permite identificar seu comportamento, que é remover e criar uma nova base de dados, sempre que a aplicação for executada.
- `Custom DB Initializer`: por meio dessa estratégia, é possível personalizar a maneira como a inicialização do `context` com a base de dados deve ocorrer.

Para utilizar uma das estratégias, é preciso registrar a utilização na classe de contexto. Adapte a classe `EFContext`, conforme mostrado a seguir.

O uso do Migrations minimiza esses problemas de atualização da base de dados. Considere seu uso em projetos futuros.

Resolução 8.12: classe que representará o contexto com a base de dados (Versão 3)

```
using System.Data.Entity;
using EFApplication.POCO;

namespace EFApplication.Contexts {
    public class EFContext : DbContext {
        public EFContext() : base("EF_Intro") {
            Database.SetInitializer<EFContext>(
                New
                DropCreateDatabaseIfModelChanges<EFContext>()
            );
        }

        public DbSet<Estado> Estados { get; set; }
    }
}
```

Execute a aplicação. Verifique que os estados anteriormente inseridos não existem mais. Caso seja do seu interesse, é possível desabilitar os inicializadores, enviando `null` como argumento para o método `SetInitializer()`. Desta maneira, não ocorrerá a perda dos dados, até que você o reconfigure.

8.10 Populando a base de dados com dados para teste

Quando realizamos testes em uma aplicação, o ideal é que ela possua um conjunto de dados disponibilizado para que operações e funcionalidades – que não sejam a inserção – possam ser testadas. Para que os dados possam ser persistidos no momento da inicialização do contexto, é preciso criar um inicializador personalizado. Assim, implemente um `Custom Initializer`, conforme apresentado a seguir.

Resolução 8.13: classe que representará o contexto com a base de dados (Versão 3)

```
public class EFInitializer :
        DropCreateDatabaseAlways<EFContext> {
    protected override void Seed(EFContext context) {
        IList<Estado> estados = new List<Estado>();
        estados.Add(new Estado()
            { UF = "PR", Nome = "Paraná" });
        estados.Add(new Estado()
            { UF = "SC", Nome = "Santa Catarina" });
        estados.Add(new Estado()
            { UF = "SP", Nome = "São Paulo" });
        estados.Add(new Estado()
            { UF = "MS", Nome = "Mato Grosso do Sul" });
        estados.Add(new Estado()
            { UF = "CE", Nome = "Ceará" });

        foreach (var estado in estados) {
            context.Estados.Add(estado);
        }

        context.SaveChanges();
        base.Seed(context);
    }
}
```

Verifique que a classe estende `DropCreateDatabaseAlways<EFContext>`, o que faz esse inicializador especializar-se para sempre remover e criar a base de dados. Essa estratégia foi adotada aqui, mas poderia ser outra, conforme a necessidade.

Nesta classe, há também a sobrescrita do método `Seed()`, para que dados para testes possam ser inseridos na base de dados e estejam disponíveis na aplicação. A compreensão é básica: declara-se uma coleção, insere-se nela os objetos que precisam ser persistidos e, em seguida, é realizada uma iteração por meio do `foreach()`, para inserir os objetos em `Estados`, que pertence ao contexto com a base de dados. Ao final, as alterações (inserções) são salvas.

É preciso agora atualizar a classe de contexto para que faça uso desse inicializador. Aliás, para finalizar a parte do modelo de negócio mapeado pelo EF, devemos inserir a propriedade `Cidades`. Tudo isso é apresentado na sequência.

Resolução 8.14: classe que representará o contexto com a base de dados (Versão 4)

```
using System.Data.Entity;
using EFApplication.POCO;

namespace EFApplication.Contexts {
    public class EFContext : DbContext{
        public EFContext() : base("EF_Intro") {
            Database.SetInitializer(
                new EFInitializer()
            );
        }

        public DbSet<Estado> Estados { get; set; }
        public DbSet<Cidade> Cidades { get; set; }
    }
}
```

8.11 Criando a interface com o usuário para aplicar a associação

A figura 8.12 apresenta a janela para a inserção de dados referente a cidades.

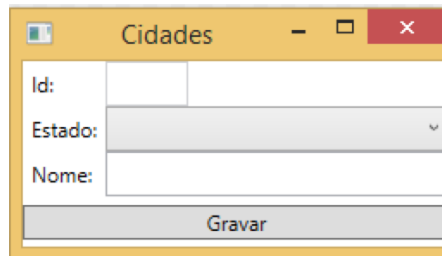



Fig. 8.12: Erro de validação de modelo encontrado pelo Entity Framework

A implementação visual para a janela `Cidades` é representada na sequência.

Resolução 8.15: XAML para a janela Cidades

```
<Window x:Class="EFApplication.Cidades"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Cidades" Height="146.448" Width="266.672"
        WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="200" />
        </Grid.ColumnDefinitions>
```

```
        <Label Grid.Row="0" Grid.Column="0" Content="Id:"/>
        <Label Grid.Row="1" Grid.Column="0"
            Content="Estado:"/>
        <Label Grid.Row="2" Grid.Column="0" Content="Nome:"/>
        <TextBox x:Name="txtID" Grid.Column="1" Grid.Row="0"
            IsEnabled="False" Margin="0,0,152,0" />
        <ComboBox x:Name="cbxEstados" Grid.Row="1"
            Grid.Column="1"/>
        <TextBox x:Name="txtNome" Grid.Column="1"
            Grid.Row="2"  />
        <Grid Grid.Row="3" Grid.ColumnSpan="2">
            <Grid.RowDefinitions>
                <RowDefinition Height="5" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Button x:Name="btnGravar" Grid.Row="1"
                Content="Gravar" />
        </Grid>
    </Grid>
</Window>
```

Com a janela desenhada com um `ComboBox` para apresentar os estados disponíveis, é preciso implementar o método para retornar os estados que o popularão. Veja o código a seguir.

Resolução 8.16: implementação do método que recupera os estados persistidos

```
private IList<Estado> GetEstados() {
    using (var context = new EFContext()) {
        return context.Estados.OrderBy(
            estado => estado.Nome).ToList<Estado>();
    }
}
```

Observe que, em relação ao método de mesmo nome para a janela `Estados`, a diferença desse método está na classificação dos estados (por nome), que, a princípio, poderia também ter sido realizada na janela de `Estados`. Desta maneira, é interessante pensar em uma reutilização, por

meio de uma classe `DAL` (*Data Access Layer*). Na sequência, é preciso implementar um método que popule o `ComboBox`. Veja tudo isso a seguir.

Resolução 8.17: implementação do método que popula e configura o ComboBox

```
private void PopulateComboBoxEstados() {
    cbxEstados.ItemsSource = GetEstados();
    cbxEstados.DisplayMemberPath = "Nome";
    cbxEstados.SelectedValuePath = "Id";
}
```

A maneira como foi apresentada a configuração de `ComboBox` em `Windows Forms` é semelhante àquela quando feita em WPF, alterando apenas o nome das propriedades. A propriedade `ItemsSource`, tal qual no `DataGrid`, é responsável por manter os itens que popularão o `ComboBox`.

Resta agora invocar o método `PopulateComboBoxEstados()` no construtor. Veja o código na sequência.

Resolução 8.18: método construtor com chamada ao método PopulateComboBoxEstados()

```
public partial class Cidades : Window {
    public Cidades() {
        InitializeComponent();
        PopulateComboBoxEstados();
    }
// Código omitido
}
```

Com a disponibilidade dos estados no `ComboBox`, é preciso implementar o processo de persistência dos dados referentes à `Cidade` que será registrada. Veja o código a seguir.

Resolução 8.19: método que persiste uma cidade

```
private Cidade SaveCidade(Cidade cidade) {
    using (var context = new EFContext()) {
```

```
        context.Cidades.Add(cidade);
        context.SaveChanges();
    }
    return cidade;
}
```

Finalizando esta funcionalidade, é preciso implementar o método que captura o evento `Click` do botão `Gravar`. Veja a seguir.

Resolução 8.20: método que captura o evento Click do botão Gravar

```
private void btnGravar_Click(object sender,
        RoutedEventArgs e){
    var cidade = SaveCidade(new Cidade() {
        EstadoId = (long) cbxEstados.SelectedValue,
        Nome = txtNome.Text
    });
    txtID.Text = cidade.Id.ToString();
}
```

Execute a aplicação, insira uma cidade e, depois, confirme a inserção no `Server Explorer`.

8.12 Aplicando Binding de objetos em um CRUD

Os primeiros passos para a implementação das operações relativas a um CRUD são: a inserção de botões que possam implementar as funcionalidades; um `DataGrid`, para que os dados persistidos possam ser selecionados; e a alimentação de dados para testes, no caso, novas cidades.

A seguir, apresento uma nova versão para o método `Seed()` da classe `EFInitializer`, onde objetos que representam as cidades são inseridos na base de dados.

Resolução 8.21: classe que representa um Initializer personalizado para o contexto e a estratégia de sempre remover e criar a base de dados (Versão 2)

```
public class EFInitializer :
        DropCreateDatabaseAlways<EFContext> {
    protected override void Seed(EFContext context) {
        IList<Estado> estados = new List<Estado>();
        estados.Add(new Estado()
            { UF = "PR", Nome = "Paraná" });
        estados.Add(new Estado()
            { UF = "SC", Nome = "Santa Catarina" });
        estados.Add(new Estado()
            { UF = "SP", Nome = "São Paulo" });
        estados.Add(new Estado()
            { UF = "MS", Nome = "Mato Grosso do Sul" });
        estados.Add(new Estado()
            { UF = "CE", Nome = "Ceará" });

        foreach (var estado in estados) {
            context.Estados.Add(estado);
        }

        IList<Cidade> cidades = new List<Cidade>();
        cidades.Add(new Cidade() { Estado = estados[0],
            Nome = "Foz do Iguaçu" });
        cidades.Add(new Cidade() { Estado = estados[1],
            Nome = "Blumenau" });
        cidades.Add(new Cidade() { Estado = estados[2],
            Nome = "Itapetininga" });
        cidades.Add(new Cidade() { Estado = estados[3],
            Nome = "Três Lagoas" });
        cidades.Add(new Cidade() { Estado = estados[4],
            Nome = "Fortaleza" });

        foreach (var cidade in cidades) {
            context.Cidades.Add(cidade);
        }

        context.SaveChanges();
        base.Seed(context);
    }
}
```

Em relação à nova janela, a figura 8.13 a apresenta.

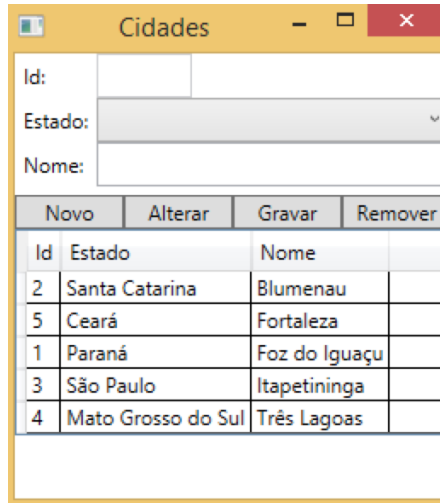



Fig. 8.13: Janela para realização das operações relativas ao CRUD de cidades

Veja na sequência a implementação para a janela da figura 8.13.

Resolução 8.22: implementação da Janela para Cidades (Versão 2)

```
<Window x:Class="EFApplication.Cidades"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Cidades" Height="300" Width="267"
        WindowStartupLocation="CenterScreen">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />

// Nova linha inserida no Data Grid
```

```
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="200" />
        </Grid.ColumnDefinitions>
        <Label Grid.Row="0" Grid.Column="0" Content="Id:"/>
        <Label Grid.Row="1" Grid.Column="0"
            Content="Estado:"/>
        <Label Grid.Row="2" Grid.Column="0" Content="Nome:"/>

// Inserida a propriedade Text nos TextBoxs e ComboBox, com
// binding no Data Grid
        <TextBox x:Name="txtID" Grid.Column="1" Grid.Row="0"
                IsEnabled="False" Margin="0,0,152,0"
                Text="{Binding SelectedItem.Id,
                ElementName=dgCidades, Mode=OneWay}"/>
            <ComboBox x:Name="cbxEstados" Grid.Row="1"
                Grid.Column="1"
                Text="{Binding SelectedItem.Estado,
                ElementName=dgCidades, Mode=OneWay}"/>
            <TextBox x:Name="txtNome" Grid.Column="1"
                Grid.Row="2" Text="{Binding SelectedItem.Nome,
                ElementName=dgCidades, Mode=OneWay}" />

// Grid que terá os botões para o CRUD
            <Grid Grid.Row="3" Grid.ColumnSpan="2">
            <Grid.RowDefinitions>
                <RowDefinition Height="5" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="64" />
                <ColumnDefinition Width="64" />
                <ColumnDefinition Width="64" />
                <ColumnDefinition Width="64" />
            </Grid.ColumnDefinitions>
            <Button x:Name="btnNovo" Grid.Row="1"
                Grid.Column="0" Content="Novo" />
```

```
            <Button x:Name="btnAlterar" Grid.Row="1"
                Grid.Column="1" Content="Alterar" />
            <Button x:Name="btnGravar" Grid.Row="1"
                Grid.Column="2" Content="Gravar" />
            <Button x:Name="btnRemover" Grid.Row="1"
                Grid.Column="3" Content="Remover"  />
        </Grid>

// Inserção do Data Grid
        <Grid Grid.Row="4" Grid.ColumnSpan="2">
            <DataGrid x:Name="dgCidades"
                ItemsSource="{Binding}"
                AutoGenerateColumns="True" IsReadOnly="True" >
            </DataGrid>
        </Grid>
    </Grid>
</Window>
```

Para popular o `DataGrid`, um método específico foi implementado (**Resolução 8.23**), semelhante ao processo anteriormente apresentado para a janela de estados. Entretanto, nas tags XAML para o `DataGrid`, a propriedade `ItemsSource` agora é definida com o valor `{Binding}`, indicando que ela estará ligada a outros controles.

Resolução 8.23: método que seleciona as cidades que serão exibidas no DataGrid

```
private IList GetCidades() {
    using (var context = new EFContext()) {
        var query = (
             from c in context.Cidades
             orderby c.Nome
             select new {
                 c.Id, c.Estado, c.Nome });
        return query.ToList();
    }
}
```

Na implementação do método `GetCidades()`, um tipo anônimo é cri-

ado na seleção dos campos que serão retornados. Para o caso de `c.Estado`, é preciso alterar o método `ToString()` da classe `Estado`, para que o nome do estado seja exibido. Veja a seguir.

Resolução 8.24: classe de negócio Estado (Versão 3)

```
public class Estado {
    public long Id { get; set; }
    public string UF { get; set; }
    public string Nome { get; set; }

    public virtual List<Cidade> Cidades { get; set; }

    public override string ToString() {
        return this.Nome;
    }
}
```

O seguinte passo é invocar o método `GetCidades()` para popular e atualizar o `DataGrid`. Veja o código na sequência.

Resolução 8.25: método que atualiza DataGrid

```
private void RefreshDataGrid() {
    dgCidades.ItemsSource = GetCidades();
}
```

A chamada ao método `RefreshDataGrid()` ocorre em dois pontos: no construtor (**Resolução 8.26**) e no método que captura o evento `Click` do botão `Gravar` (**Resolução 8.27**).

Resolução 8.26: método construtor com chamada ao método RefreshDataGrid()

```
public partial class Cidades : Window {
    public Cidades() {
        InitializeComponent();
        PopulateComboBoxEstados();
```

```
        RefreshDataGrid();
    }
// Código omitido
}
```

Resolução 8.27: método que captura o evento Click do botão Gravar com chamada ao método RefreshDataGrid()

```
private void btnGravar_Click(object sender,
        RoutedEventArgs e) {
    var cidade = SaveCidade(new Cidade() {
        EstadoId = (long) cbxEstados.SelectedValue,
        Nome = txtNome.Text
    });
    txtID.Text = cidade.Id.ToString();
    RefreshDataGrid();
}
```

Na sequência, para que os controles de entrada de dados (`TextBox`s e `ComboBox`) estejam ligados ao `DataGrid`, e que os dados dela sejam atualizados nesses controles, ao se selecionar uma cidade, é preciso realizar algumas configurações. A seguir, apresento um fragmento do código da **Resolução 8.22**, referente à janela `Cidades`.

Resolução 8.28: trecho de código de binding com DataGrid

```
// Inserida a propriedade Text nos TextBoxs e ComboBox, com
// binding no Data Grid
    <TextBox x:Name="txtID" Grid.Column="1" Grid.Row="0"
        IsEnabled="False" Margin="0,0,152,0"
        Text="{Binding SelectedItem.Id,
        ElementName=dgCidades, Mode=OneWay}"/>
    <ComboBox x:Name="cbxEstados" Grid.Row="1"
        Grid.Column="1"
        Text="{Binding SelectedItem.Estado,
        ElementName=dgCidades, Mode=OneWay}"/>
    <TextBox x:Name="txtNome" Grid.Column="1" Grid.Row="2"
        Text="{Binding SelectedItem.Nome,
        ElementName=dgCidades, Mode=OneWay}" />
```

Um controle com ligação (*binding*) com outro controle precisa tê-la definida em sua propriedade que receberá o valor dessa ligação (no caso, a `Text`). Essa ligação informa qual a propriedade que dará origem aos dados (`SelectedItem`), qual é o controle que fornecerá a ligação (`ElementName`) e qual é o seu modo (`Mode`).

O WPF fornece recursos ricos e simples para que os elementos de interação com o usuário possam exibir dados. Os controles (elementos) podem ser ligados a uma grande variedade de fontes de dados, como recursos da CLR, XML e controles etc.

Quando se ligam controles, para que os dados de um possam atualizar outros, é preciso definir em qual direção ocorre essa ligação/atualização. As atualizações de ligação podem ocorrer: apenas na propriedade de destino (*target*), apenas na propriedade de origem/fonte (*source*) ou em ambas. Para isso, é preciso atribuir o tipo de direcionamento ao atributo `Mode`, que pode ser `TwoWay`, `OneWay`, `OneTime`, `OneWayToSource` ou `Default`.

Execute a aplicação, selecione uma cidade no `DataGrid` e verifique se os controles referentes aos dados dela são atualizados. Na sequência, é preciso implementar o comportamento dos botões referentes ao CRUD. O primeiro é o botão `Novo`, que deverá "limpar" os controles, para que novos dados possam ser informados. Veja a seguir.

Resolução 8.29: método que captura o evento Click do botão Novo

```
private void btnNovo_Click(object sender, RoutedEventArgs e) {
    dgCidades.UnselectAll();
}
```

Uma vez que os dados dos controles de interação estão ligados com o `DataGrid`, ao removermos sua seleção, os controles são automaticamente "limpados", sendo possível a inserção de um novo registro. Teste esta funcionalidade.

Na sequência, é preciso implementar a remoção de uma cidade. Para isso, um método específico precisa ser criado, conforme apresento a seguir.

Resolução 8.30: método responsável pela remoção de uma cidade

```
private void RemoveCidade(long idCidade) {
    using (var context = new EFContext()) {
        var cidade = context.Cidades.Find(idCidade);
        context.Cidades.Remove(cidade);
        context.SaveChanges();
    }
}
```

Observe que, para a remoção de uma cidade, o objeto que a representa precisou ser recuperado e trazido para o contexto. Este procedimento é necessário para que o EF compreenda que o objeto enviado como argumento não é um objeto desconectado (*detached*).

Para invocar o método `RemoveCidade()`, implemente o código a seguir no método que captura o evento `Click` do botão `Remover`.

Resolução 8.31: método que captura o evento Click do botão Remover

```
private void btnRemover_Click(object sender,
        RoutedEventArgs e) {
    RemoveCidade(Convert.ToInt64(txtID.Text));
    RefreshDataGrid();
}
```

Finalizando o CRUD para cidades, é preciso implementar a funcionalidade para o botão de alteração de uma cidade. Veja a seguir.

Resolução 8.32: método para atualização de uma cidade

```
private void UpdateCidade(Cidade cidade) {
    using (var context = new EFContext()) {
        var newCidade = context.Cidades.Find(cidade.Id);
        newCidade.Nome = cidade.Nome;
        newCidade.EstadoId = cidade.EstadoId;
        context.SaveChanges();
    }
}
```

Veja no código do método `UpdateCidade()` que uma cidade é recuperada; nela são atualizadas as propriedades e não há nenhuma chamada a algum método para atualizar a nova cidade. Essa atualização é feita de maneira automática pelo EF, pois o objeto alterado pertence ao contexto com a base de dados.

A chamada a esse método é realizada no método que captura o evento `Click` do botão `Alterar`. Veja o código a seguir.

Resolução 8.33: método que captura o evento Click do botão Alterar

```
private void btnAlterar_Click(object sender,
        RoutedEventArgs e) {
    UpdateCidade(new Cidade() {
        Id = Convert.ToInt64(txtID.Text),
        Nome = txtNome.Text,
        EstadoId = (long) cbxEstados.SelectedValue
    });
    RefreshDataGrid();
}
```

8.13 Conclusão

O uso de uma ferramenta que subsidie o mapeamento automático de objetos para uma base de dados relacional – e também o caminho inverso, no qual registros são mapeados para objetos – é uma real necessidade em aplicações orientadas a objetos. O Entity Framework, apresentado neste capítulo, permite que a equipe de desenvolvimento concentre-se na lógica de negócio, não se preocupando em como os dados serão armazenados e/ou recuperados. Tampouco há a mescla de código OO com instruções SQL.

O conteúdo apresentado sobre o EF neste capítulo não esgotou os recursos, características e técnicas possíveis de serem aplicados. Os exemplos limitaram-se a apresentar o mapeamento por convenção, onde nenhum código adicional foi implementado nas classes. Caso alguma configuração seja necessária, é possível customizar a maneira como o EF visualizará a classe de negócio, em relação à base de dados, fazendo uso de *Data Annotations*.

Com vistas para a aplicação do EF, foi usado novamente o Windows Presentation Foundation, trazendo também novos recursos, como o uso de `ComboBox` e de ligações entre controles.

Este capítulo finaliza a proposta do livro. Agora fica para você o desafio de aplicar todo o conhecimento adquirido em suas aplicações futuras.

Capítulo 9
Os estudos não param por aqui

O .NET já não é uma novidade, é uma plataforma estável que vem ganhando cada vez mais adeptos e que trouxe uma nova e poderosa linguagem, o C#, com uma curva de aprendizado relativamente boa. Um dos recursos oferecidos por ele é o desenvolvimento de aplicações para desktop, sendo este o objetivo deste livro cuja leitura você acabou de concluir, usando constantemente o C# em todos os capítulos. Para o desenvolvimento deste tipo de aplicativo (e de todos os oferecidos pelo .NET), a Microsoft disponibilizou o Visual Studio, que tem versões pagas e gratuitas.

Para que você pudesse se ambientar de maneira gradativa com a plataforma, a linguagem e o ambiente de desenvolvimento, a técnica usada nos primeiros capítulos foi de apresentar resoluções básicas, fazendo uso das principais estruturas de implementação de algoritmos: sequencial, condicional e de repetição. Como C# é uma linguagem totalmente orientada a objetos, conceitos deste paradigma também foram apresentados, todos com aplicações que

permitiram suas implementações.

As tecnologias oferecidas pelo .NET são diversas e algumas delas foram apresentadas durante o livro. Para acesso à base de dados, a plataforma oferece o ADO.NET, que possui uma hierarquia de classes, possibilitando uma conexão, consulta e atualização. Dentro destas classes, existe o `DataSet`, que permite trazer a estrutura de uma base de dados para a sua aplicação e trabalhar com ela localmente, na memória da máquina do cliente.

Outra tecnologia vista foi o LINQ, que permite consultas a coleções de dados, sejam elas baseadas na API Collections ou em uma base de dados. Foi apresentado o Entity Frameworks (EF), um poderoso gerenciador de mapeamento de objetos para uma base relacional (e vice-versa) – conhecido também como ORM –, como também um framework para desenvolvimento de aplicações desktop, o Windows Presentation Foundation (WPF), utilizado também de forma introdutória neste livro.

Espera-se que com os recursos e técnicas exibidos, alguns tenham provocado em você uma curiosidade ou interesse em um aprofundamento, o que certamente agora se tornará mais fácil. Programar com C# é muito bom, e criar aplicações usando .NET não é difícil.

Comece agora a criar suas próprias aplicações. Quando surgirem dificuldades, você verá que a comunidade existente na internet, disposta a lhe auxiliar, é muito grande. Vale relembrar do grupo de discussão deste livro em específico: https://groups.google.com/forum/#!forum/livro-windows-forms-casa-do-codigo.

Sucesso, e que a força esteja com você! ;-)

Índice Remissivo